FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

# Integration and application of symbolic and automatic differentiation frameworks into deal.II

Jean-Paul Pelteret, Isuru Fernando, Andrew McBride, Paul Steinmann

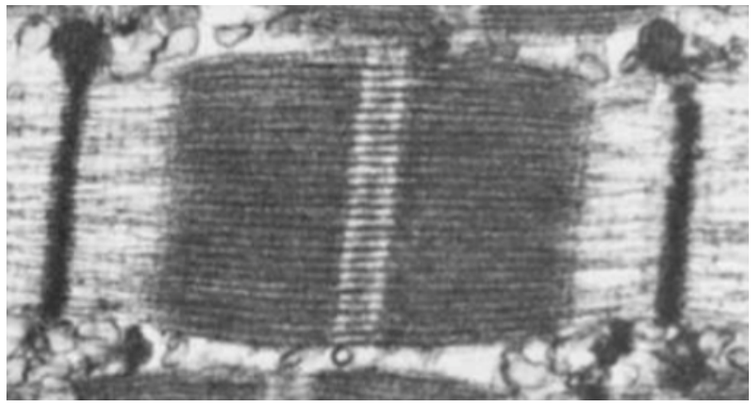
Lehrstuhl für Technische Mechanik
Friedrich-Alexander Universität Erlangen-Nürnberg

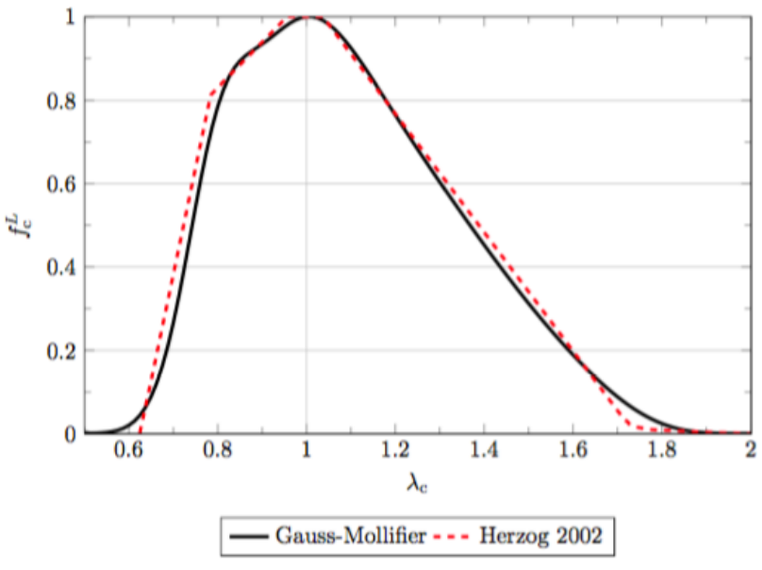
24 July 2018

# Motivation

## Examples of material instability phenomenon in coupled media

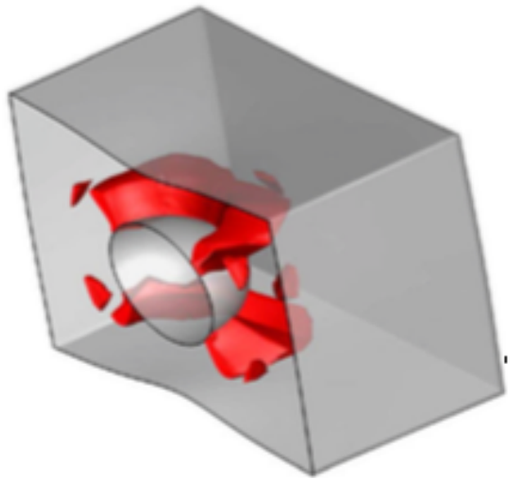
### Active skeletal muscle

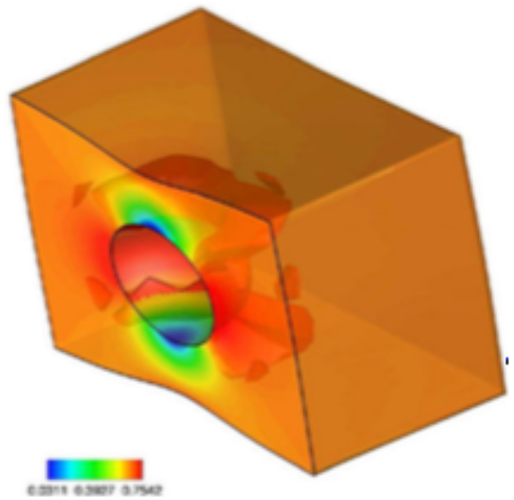





Hagopian (1970), Pelteret (2013)

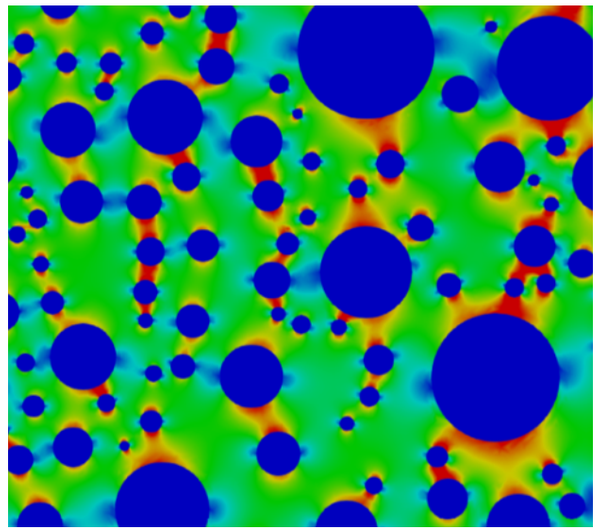
### EEAPs



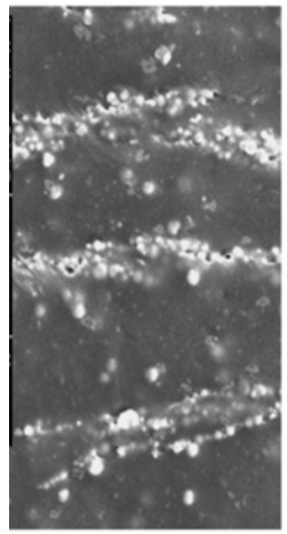


Miehe (2015), Rudykh (2013a)

### MREs



Pelteret (2016)



Rudykh (2013b)

- Soils, concrete, rocks, ceramics, metals, polymers, composites, etc… de Borst (1998)

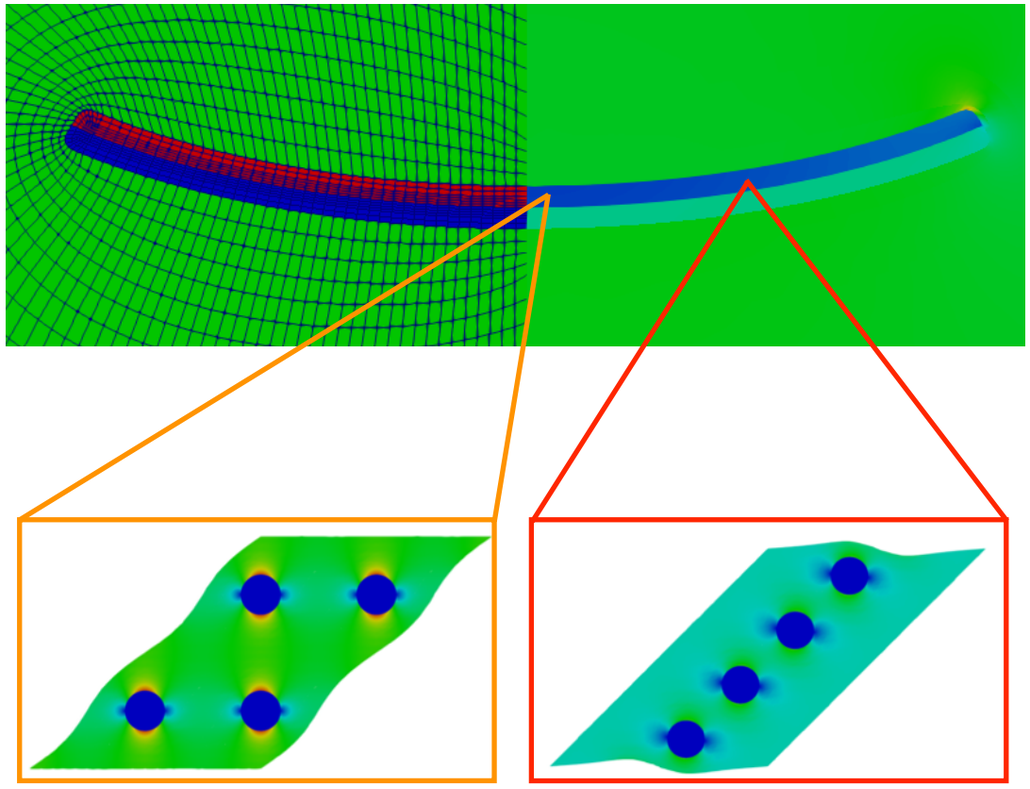
# Motivation

**Implementing sophisticated computational models correctly is a challenging task!**

**Errors in linearisation may be misconstrued as physically plausible material instabilities.**

Desire a toolset to help with these issues

- Validation
  - (Intricate) Constitutive models
  - Linearisation of finite element (FE) residuals
  - Unstable formulations (non-convex material models)
- Rapid prototyping
  - New constitutive models and FE formulations
  - All tasks more tedious in a multi-physics framework
  - Important to keep in mind:
    - For many academic applications, most “research hours” spent in development rather than actual simulation time



# Presentation outline

- Automatic differentiation
  - Introduction
  - Framework in deal.II
- Symbolic differentiation
  - Introduction
  - Framework in deal.II
- Application to magneto-mechanical problems
  - Constitutive modelling
  - Finite element modelling

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

# Automatic differentiation

deal.II

# Automatic differentiation

## What is AD?

- A numerical method that can be used to "automatically" compute the first, and perhaps higher-order, derivatives of function(s) with respect to one or more input variables.

$$\boldsymbol{f}(\boldsymbol{x})|_{\boldsymbol{x}=\bar{\boldsymbol{x}}} \quad \rightarrow \quad \boldsymbol{f}'^{(n)}(\boldsymbol{x})|_{\boldsymbol{x}=\bar{\boldsymbol{x}}}$$

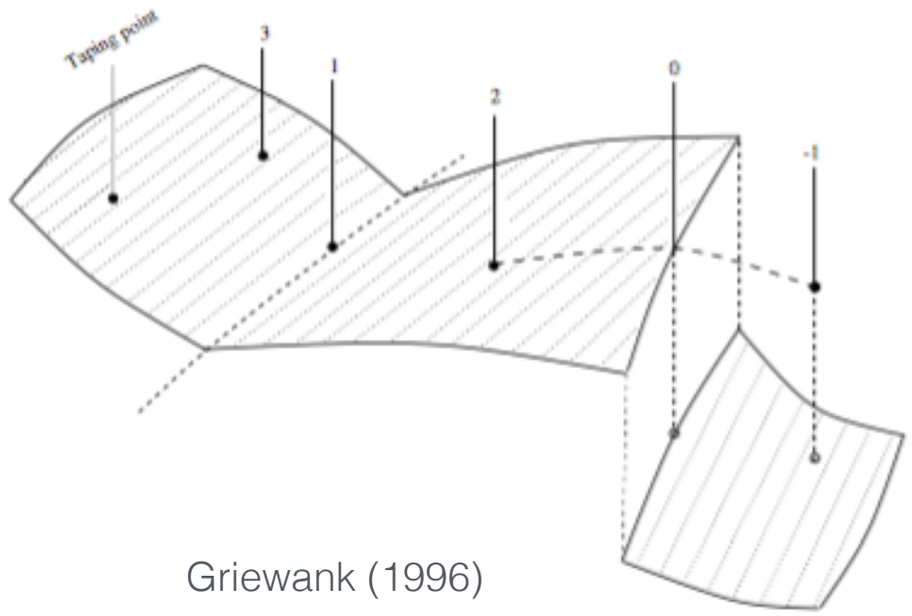
- Can be accurate to machine precision
- Different to numerical differentiation
  - Errors due to Taylor series truncation and step-size

# Automatic differentiation

## Categorisation of AD frameworks

- Source code transformation (generation)
- Operator overloading
- taping strategies
- dual / complex-step / hyper-dual numbers (tapeless)
- expression templates (compile time operation)
- forward / reverse mode

$$f(\mathbf{x}) = f_0 \circ f_1 \circ f_2 \circ \ldots \circ f_n(\mathbf{x})$$




Griewank (1996)

$$f(x+d) = f(x) + df'(x) + \frac{1}{2!}d^2 f''(x) + \frac{1}{3!}d^3 f'''(x) + \ldots$$

Fike (2011)

FAD

$$\frac{df(\mathbf{x})}{d\mathbf{x}} = \frac{df_0}{df_1}\left(\frac{df_1}{df_2}\left(\frac{df_2}{df_3}\left(\ldots\left(\frac{df_n(\mathbf{x})}{d\mathbf{x}}\right)\right)\right)\right)$$

$$\frac{df(\mathbf{x})}{d\mathbf{x}} = \left(\left(\left(\left(\left(\frac{df_0}{df_1}\right)\frac{df_1}{df_2}\right)\frac{df_2}{df_3}\right)\ldots\right)\frac{df_n(\mathbf{x})}{d\mathbf{x}}\right)$$

RAD

# Integration of AD into deal.II library

## Compatibility with existing functions and classes

- Pertinent deal.II classes already templated on number type
- Linear Algebra: **`Tensor`**, **`SymmetricTensor`** classes
  - Challenge 1: Ensuring correct/safe initialisation while maintaining compatibility with existing code
  - Challenge 2: Ensuring a valid return types for numbers based on expression templates
  - Challenge 3: Ensuring a valid return types for temporary, non-copyable numbers
- Integrator: **`FEValues`** class
  - Challenge: Identifying shortcuts that reduce computational expense but invalidate differentiability
- Eigenvalue/vector solver for symmetric tensors
  - Challenge: Ensuring differentiability of results based on diagonal tensors

```
template <int rank, int dim,
          typename Number>
class Tensor;
```

Tensor operators with template constraints

```
template <int rank, int dim,
          typename Number,
          typename OtherNumber>
Tensor<rank,dim,typename ProductType<
       typename EnableIfScalar<Number>::type,
       OtherNumber
      >::type>
operator * (const Number                    &factor,
            const Tensor<rank,dim,OtherNumber> &t);
```

# Integration of AD into deal.II library

## Drivers: A compatibility layer for AD libraries

- Supported AD libraries:
  - ADOL-C (taped, tapeless)
    Griewank (1996)
  - Sacado (combinations of DFad, Rad)
    Gay (2012), Phipps (2012)
- Each library has a different interface / works differently
  - Underlying scalar values in operations
  - Marking independent and dependent variables
  - Performing differentiation
  - Value extraction

ADOL-C (taped)

```
const unsigned int n_ind = 10;
std::vector<double> x (n_ind);
std::vector<::adouble> x_ad (n_ind);

for (unsigned int i=0; i<n_ind; i++) {
  x[i] = 2*i;
  x_ad[i] <<= x[i];
}
```

Sacado (DFad)

```
const unsigned int n_ind = 10;
std::vector<double> x (n_ind);
std::vector<Sacado::Fad::DFad<double>> x_ad (n_ind);

for (unsigned int i=0; i<n_ind; i++) {
  x[i] = 2*i;
  x_ad[i] = Sacado::Fad::DFad<double>(n_ind,i,x[i]);
}
```

deal.II

```
const unsigned int n_ind = 10;
dealii::ADHelper ad_helper (n_ind);

std::vector<double> x (n_ind);
for (unsigned int i=0; i<n_ind; i++)
  x[i] = 2*i;

ad_helper.register_independent_variables(x);
// Calls internal::Marking<ad_type>::independent_variable(...)
```

# Integration of AD into deal.II library

## Differentiation helpers

- QP-level scalar function
  (accounts for symmetries of ***A***)

$$\Psi\left(\boldsymbol{A}\right) \quad \rightarrow \quad \frac{d\Psi\left(\boldsymbol{A}\right)}{d\boldsymbol{A}}, \frac{d^2\Psi\left(\boldsymbol{A}\right)}{d\boldsymbol{A} \otimes d\boldsymbol{A}}$$

- QP-level vector function
  (accounts for symmetries of ***A***)

$$\boldsymbol{S}\left(\boldsymbol{A}\right) \quad \rightarrow \quad \frac{d\boldsymbol{S}\left(\boldsymbol{A}\right)}{d\boldsymbol{A}}$$

- Cell-level energy functional

$$\Pi\left(\boldsymbol{d}\right) \quad \rightarrow \quad \boldsymbol{R} = -\frac{d\,\Pi\left(\boldsymbol{d}\right)}{d\boldsymbol{d}}, \boldsymbol{K} = \frac{d^2\Pi\left(\boldsymbol{d}\right)}{d\boldsymbol{d} \otimes d\boldsymbol{d}}$$

- Cell-level residual linearisation

$$\boldsymbol{R}\left(\boldsymbol{d}\right) \quad \rightarrow \quad \boldsymbol{K} = -\frac{d\boldsymbol{R}\left(\boldsymbol{d}\right)}{d\boldsymbol{d}}$$

```
template<int dim, enum AD::NumberTypes ADNumberTypeCode, typename ScalarType>
class ADHelperVariationalFormulation : ...
{
public:
 ADHelperVariationalFormulation(const unsigned int n_independent_variables);

 // Independent variables
 void register_dof_values (const std::vector<scalar_type> &dof_values);
 const std::vector<ad_type>& get_sensitive_dof_values ();

 // Dependent variables
 void register_energy_functional (const ad_type &energy);

 // Localised assembly
 Vector<scalar_type> compute_residual() const override;
 FullMatrix<scalar_type> compute_linearization() const override;
};
```

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

# Automatic differentiation example

## Differentiating a multi-field scalar function

$$\Psi(s, \mathbf{v}, \mathbf{T}) = s^{2.2} [\mathbf{v} \cdot \mathbf{v}]^3 [\det(\mathbf{T})]^2$$

deal.II

# Integration of AD into deal.II library

## Defining energy function; preliminary details

```
template<int dim, typename NumberType>
NumberType
psi (const Tensor<2,dim,NumberType> &t, const Tensor<1,dim,NumberType> &v, const NumberType &s)
{
  return std::pow(determinant(t),2)*std::pow(v*v,3)*std::pow(s,sf);
};
```

$$\Psi(s, \mathbf{v}, \mathbf{T}) = s^{2.2} [\mathbf{v} \cdot \mathbf{v}]^3 [\det(\mathbf{T})]^2$$

```
template<int dim, typename number_t, enum AD::NumberTypes ad_type_code>
void test_tensor_vector_scalar_coupled ()
{
  // Choose an AD helper that expects one dependent variable
  typedef AD::ADHelperScalarFunction<dim,ad_type_code,number_t> ADHelper;
  typedef typename ADHelper::ad_type     ADNumberType;
  typedef typename ADHelper::scalar_type ScalarNumberType;

  // Initialise values for independent variables
  ScalarNumberType                s = ...;
  Tensor<1,dim,ScalarNumberType> v = ...;
  Tensor<2,dim,ScalarNumberType> t = ...;

  // Setup local indexing for independent variables
  const FEValuesExtractors::Tensor<2> t_dof (0);
  const FEValuesExtractors::Vector    v_dof (Tensor<2,dim>::n_independent_components);
  const FEValuesExtractors::Scalar    s_dof (Tensor<2,dim>::n_independent_components
                                             + Tensor<1,dim>::n_independent_components);
```

# Integration of AD into deal.II library

## Defining energy function; preliminary details

```
template<int dim, typename NumberType>
NumberType
psi (const Tensor<2,dim,NumberType> &t, const Tensor<1,dim,NumberType> &v, const NumberType &s)
{
  return std::pow(determinant(t),2)*std::pow(v*v,3)*std::pow(s,sf);
};
```

$$\Psi(s, \mathbf{v}, \mathbf{T}) = s^{2.2} [\mathbf{v} \cdot \mathbf{v}]^3 [\det(\mathbf{T})]^2$$

```
template<int dim, typename number_t, enum AD::NumberTypes ad_type_code>
void test_tensor_vector_scalar_coupled ()
{
  // Choose an AD helper that expects one dependent variable
  typedef AD::ADHelperScalarFunction<dim,ad_type_code,number_t> ADHelper;
  typedef typename ADHelper::ad_type     ADNumberType;
  typedef typename ADHelper::scalar_type ScalarNumberType;

  // Initialise values for independent variables
  ScalarNumberType                s = ...;
  Tensor<1,dim,ScalarNumberType> v = ...;
  Tensor<2,dim,ScalarNumberType> t = ...;

  // Setup local indexing for independent variables
  const FEValuesExtractors::Tensor<2> t_dof (0);
  const FEValuesExtractors::Vector    v_dof (Tensor<2,dim>::n_independent_components);
  const FEValuesExtractors::Scalar    s_dof (Tensor<2,dim>::n_independent_components
                                             + Tensor<1,dim>::n_independent_components);
```

# Integration of AD into deal.II library

## Configuring helper; registering the function to be differentiated

```
// Setup AD helper
const unsigned int n_AD_components = Tensor<2,dim>::n_independent_components
                                   + Tensor<1,dim>::n_independent_components
                                   + 1;
ADHelper ad_helper (n_AD_components);

// Start tracing operations
const bool is_recording = ad_helper.enable_record_sensitivities(tape_no);
if (is_recording) {
  // Set values of independents and get sensitive counterparts
  // Same process applies for other variables s,v
  ad_helper.register_independent_variable(t, t_dof);
  const Tensor<2,dim,ADNumberType> t_ad = ad_helper.get_sensitive_variables(t_dof);

  // Compute function value
  const ADNumberType psi_ad (psi(t_ad,v_ad,s_ad));

  // Register the dependent function
  ad_helper.register_dependent_variable(psi_ad);

  ad_helper.disable_record_sensitivities();
}
else {
 ad_helper.activate_tape(tape_no);
 // Change values of independent variables
 t = ...;
 ad_helper.set_independent_variable(t, t_dof); // Same for other variables s,v
}
```

# Integration of AD into deal.II library

## Configuring helper; registering the function to be differentiated

```
// Setup AD helper
const unsigned int n_AD_components = Tensor<2,dim>::n_independent_components
                                   + Tensor<1,dim>::n_independent_components
                                   + 1;
ADHelper ad_helper (n_AD_components);

// Start tracing operations
const bool is_recording = ad_helper.enable_record_sensitivities(tape_no);
if (is_recording) {
  // Set values of independents and get sensitive counterparts
  // Same process applies for other variables s,v
  ad_helper.register_independent_variable(t, t_dof);
  const Tensor<2,dim,ADNumberType> t_ad = ad_helper.get_sensitive_variables(t_dof);

  // Compute function value
  const ADNumberType psi_ad (psi(t_ad,v_ad,s_ad));

  // Register the dependent function
  ad_helper.register_dependent_variable(psi_ad);

  ad_helper.disable_record_sensitivities();
}
else {
 ad_helper.activate_tape(tape_no);
 // Change values of independent variables
 t = ...;
 ad_helper.set_independent_variable(t, t_dof); // Same for other variables s,v
}
```

# Integration of AD into deal.II library

## Configuring helper; registering the function to be differentiated

```
// Setup AD helper
const unsigned int n_AD_components = Tensor<2,dim>::n_independent_components
                                   + Tensor<1,dim>::n_independent_components
                                   + 1;
ADHelper ad_helper (n_AD_components);

// Start tracing operations
const bool is_recording = ad_helper.enable_record_sensitivities(tape_no);
if (is_recording) {
  // Set values of independents and get sensitive counterparts
  // Same process applies for other variables s,v
  ad_helper.register_independent_variable(t, t_dof);
  const Tensor<2,dim,ADNumberType> t_ad = ad_helper.get_sensitive_variables(t_dof);

  // Compute function value
  const ADNumberType psi_ad (psi(t_ad,v_ad,s_ad));

  // Register the dependent function
  ad_helper.register_dependent_variable(psi_ad);

  ad_helper.disable_record_sensitivities();
}
else {
 ad_helper.activate_tape(tape_no);
 // Change values of independent variables
 t = ...;
 ad_helper.set_independent_variable(t, t_dof); // Same for other variables s,v
}
```

# Integration of AD into deal.II library

## Performing differentiation; extracting derivative components

```
// Compute the function value, gradient and hessian for the (new) evaluation point
const ScalarNumberType psi_sclr = ad_helper.compute_value();
const Vector<ScalarNumberType> Dpsi = ad_helper.compute_gradient();

FullMatrix<ScalarNumberType> D2psi;
if (AD::ADNumberTraits<ADNumberType>::n_supported_derivative_levels >= 2) {
  D2psi = ad_helper.compute_hessian();
}

// Extract components of the solution total derivatives
const Tensor<2,dim,ScalarNumberType> dpsi_dt
  = ad_helper.extract_gradient_component(Dpsi,t_dof);

if (AD::ADNumberTraits<ADNumberType>::n_supported_derivative_levels >= 2) {
  const Tensor<4,dim,ScalarNumberType> d2psi_dt_dt
    = ad_helper.extract_hessian_component(D2psi,t_dof,t_dof);
  const Tensor<3,dim,ScalarNumberType> d2psi_dv_dt
    = ad_helper.extract_hessian_component(D2psi,t_dof,v_dof);
}
}
```

# Integration of AD into deal.II library

## Performing differentiation; extracting derivative components

```
// Compute the function value, gradient and hessian for the (new) evaluation point
const ScalarNumberType psi_sclr = ad_helper.compute_value();
const Vector<ScalarNumberType> Dpsi = ad_helper.compute_gradient();

FullMatrix<ScalarNumberType> D2psi;
  if (AD::ADNumberTraits<ADNumberType>::n_supported_derivative_levels >= 2) {
    D2psi = ad_helper.compute_hessian();
  }

  // Extract components of the solution total derivatives
  const Tensor<2,dim,ScalarNumberType> dpsi_dt
    = ad_helper.extract_gradient_component(Dpsi,t_dof);

  if (AD::ADNumberTraits<ADNumberType>::n_supported_derivative_levels >= 2) {
    const Tensor<4,dim,ScalarNumberType> d2psi_dt_dt
      = ad_helper.extract_hessian_component(D2psi,t_dof,t_dof);
    const Tensor<3,dim,ScalarNumberType> d2psi_dv_dt
      = ad_helper.extract_hessian_component(D2psi,t_dof,v_dof);
  }
}
```

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

# Symbolic differentiation

deal.II

# Symbolic differentiation

## What is SD?

- A Computer Algebra System (CAS) for manipulating algebraic expressions

$$f(x, y(x, z)) = 3x + y^2 \quad , \quad y = x/z$$

  - Not limited to differentiation, but can also perform other mathematical operations (e.g. integration, factorisation etc.)

  - Full control over interpretation of symbolic expressions

    - Numerical values

    - Symbolic functions

    - Other constructs

- Traditionally considered to be “slow” (in comparison to AD, for example)

## SymEngine library

Meurer (2017)

- Symbolic manipulation library

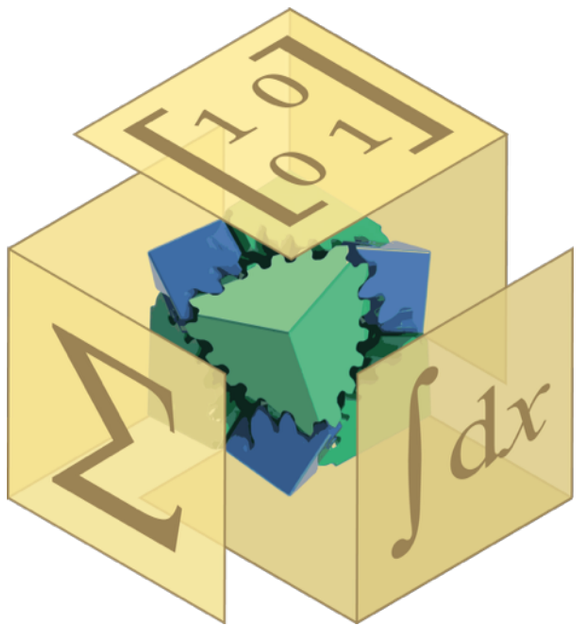
  - New backend for SymPy

  - Wrappers for numerous languages

- Written in (modern) C++

  - Performance orientated

  - Visitor pattern



# Integration of SD into deal.II library

## Compatibility with existing functions and classes

- Pertinent deal.II classes already templated on number type
- **`SymEngineNumber`** class to provide operator-overloaded interface to SymEngine

```
class SymEngineNumber
{
private:
 // Symbolic number, variable or function
 SE::RCP<const SE::Basic> value;

public:

 // Constructors
 template<typename NumberType>
 explicit SymEngineNumber(const NumberType &val);

 SymEngineNumber(const std::string &sym);

 // Operators
 SymEngineNumber &
 operator+=(const SymEngineNumber &rhs)
 {
   value = SE::add(value, rhs.value);
   return *this;
 }

 // Differentiation
 SymEngineNumber
 derivative(const SymEngineNumber &symbol) const;

 // Substitution (partial / full)
 template<typename NumberType>
 SymEngineNumber
 substitution(const SymEngineNumber &symbol,
              const NumberType      &value) const;

 SymEngineNumber
 substitution(
  const typename SymEngineNumber::substitution_map_t
  &substitution_values) const;

 // Evaluation and casting
 template<typename ReturnType>
 ReturnType substitution_with_evaluation(
  const typename SymEngineNumber::substitution_map_t
  &substitution_values) const;

 template <typename ResultType>
 explicit operator ResultType() const;
};
```

# Integration of SD into deal.II library

## Convenience functions

- Interaction with symbolic equivalents of commonly used linear algebra classes (scalar types, **`Tensor`**, **`SymmetricTensor`**)
  - Initialisation
  - Tensor differentiation (arbitrary order, accounts for symmetries)

$$\begin{array}{rcl} \boldsymbol{S}(\boldsymbol{A},\boldsymbol{B}) & \rightarrow & \dfrac{\partial \boldsymbol{S}(\boldsymbol{A},\boldsymbol{B})}{\partial \boldsymbol{A}}\Big|_{\boldsymbol{B}} \\ \downarrow d/d\mathbf{A} & & \\ \boldsymbol{S}(\boldsymbol{A},\boldsymbol{B}(\boldsymbol{A})) & \rightarrow & \dfrac{\partial \boldsymbol{S}(\boldsymbol{A},\boldsymbol{B}(\boldsymbol{A}))}{\partial \boldsymbol{A}}\Big|_{\boldsymbol{B}} + \dfrac{\partial \boldsymbol{S}(\boldsymbol{A},\boldsymbol{B}(\boldsymbol{A}))}{\partial \boldsymbol{B}}\Big|_{\boldsymbol{A}} \dfrac{d\boldsymbol{B}(\boldsymbol{A})}{d\boldsymbol{A}} \end{array}$$

```
typedef SD::SymEngineNumber SDNumberType;

// Initialisation
const Tensor<2,dim,SDNumberType> symb_t (
  SD::make_symbol_tensor<2,dim>("t"));
```

```
const SDNumberType symb_psi = psi(symb_t);

// Differentiation
const Tensor<2,dim,SDNumberType> symb_dpsi_dt
  = SD::differentiate(symb_psi,symb_t);

const Tensor<4,dim,SDNumberType> symb_d2psi_dt_dt
  = SD::differentiate(symb_dpsi_dt,symb_t);
```

- Substitution maps
  - Creation (scalar, tensor independent variables)
  - Partial substitution (scalar, tensor dependent variables)
  - Resolution of explicit dependencies

Incomplete substitution

$$f(a,b(a)) = a + b$$
$$\rightarrow \quad f|_{a=1,b=a+2} = 3 + a$$

Complete substitution

$$f(a(b),b) = a + b$$
$$\rightarrow \quad f|_{a=b+2,b=1} = [b + 2 + b]_{b=1} = 4$$

# Integration of SD into deal.II library

## Convenience functions

- Interaction with symbolic equivalents of commonly used linear algebra classes (scalar types, **`Tensor`**, **`SymmetricTensor`**)
  - Initialisation
  - Tensor differentiation (arbitrary order, accounts for symmetries)

$$\begin{array}{rcl} \boldsymbol{S}(\boldsymbol{A},\boldsymbol{B}) & \rightarrow & \dfrac{\partial \boldsymbol{S}(\boldsymbol{A},\boldsymbol{B})}{\partial \boldsymbol{A}}\Big|_{\boldsymbol{B}} \\ \quad\quad d/d\mathbf{A} & & \\ \boldsymbol{S}(\boldsymbol{A},\boldsymbol{B}(\boldsymbol{A})) & \rightarrow & \dfrac{\partial \boldsymbol{S}(\boldsymbol{A},\boldsymbol{B}(\boldsymbol{A}))}{\partial \boldsymbol{A}}\Big|_{\boldsymbol{B}} + \dfrac{\partial \boldsymbol{S}(\boldsymbol{A},\boldsymbol{B}(\boldsymbol{A}))}{\partial \boldsymbol{B}}\Big|_{\boldsymbol{A}} \dfrac{d\boldsymbol{B}(\boldsymbol{A})}{d\boldsymbol{A}} \end{array}$$

  - Substitution maps
    - Creation (scalar, tensor independent variables)
    - Partial substitution (scalar, tensor dependent variables)
    - Resolution of explicit dependencies

```
typedef SD::SymEngineNumber SDNumberType;

// Initialisation
const Tensor<2,dim,SDNumberType> symb_t (
  SD::make_symbol_tensor<2,dim>("t"));
```

```
const SDNumberType symb_psi = psi(symb_t);

// Differentiation
const Tensor<2,dim,SDNumberType> symb_dpsi_dt
  = SD::differentiate(symb_psi,symb_t);

const Tensor<4,dim,SDNumberType> symb_d2psi_dt_dt
  = SD::differentiate(symb_dpsi_dt,symb_t);
```

### Incomplete substitution

$$f(a,b(a)) = a + b$$

$$\rightarrow \quad f|_{a=1,b=a+2} = 3 + a$$

### Complete substitution

$$f(a(b),b) = a + b$$

$$\rightarrow \quad f|_{a=b+2,b=1} = [b + 2 + b]_{b=1} = 4$$

# Integration of SD into deal.II library

## Optimisation

- Common subexpression elimination (CSE)
- “lambda”: String to function transformation

```
std::function<NumberType(
   const NumberType *values)> func;
```

- LLVM: Just in time (JIT) compilation to LLVM byte-code
  - Aggressive optimisation for vectorised expressions

## AD numbers as custom types

- Linearisation using AD instead of SD
- Wrappers for “lambda” optimiser, CSE
  - Taped AD types:
    Shared memory pool minimises active variable count

## LinearOperators as custom types

- Define solvers, preconditioners at run-time using a parameter file

```
// Create a symbolic expression to be evaluated
const SDNumberType symb_lo_A_inv("lo_A_inv");
const SDNumberType symb_la_u("la_u");
const SDNumberType symb_la_v("la_v");
const SDNumberType symb_la_y
  = symb_lo_A_inv * (symb_la_u + symb_la_v);

// Setup a linear system
SparseMatrix<number_t> la_A(...);
Vector<number_t>       la_u(...);
Vector<number_t>       la_v(...);

// Create linear operators
LinearOperator<Vector<number_t>> lo_A(la_A);
LinearOperator<Vector<number_t>> lo_A_inv
  = inverse_operator(lo_A, ...);

// Create a substitution map, linking the symbolic expression
// to the linear algebra
SDNumberType::substitution_map_t sub_vals;
sub_vals[symb_lo_A_inv] = SDNumberType(lo_A_inv, "A_inv");
sub_vals[symb_la_u]     = SDNumberType(la_u, "u");
sub_vals[symb_la_v]     = SDNumberType(la_v, "v");

// Substitution with evaluation
const Vector<number_t> la_y_result =
  static_cast<Vector<number_t>>(symb_la_y.substitution(sub_vals));
```

# Integration of SD into deal.II library

## Optimisation

- Common subexpression elimination (CSE)
- “lambda”: String to function transformation

```
std::function<NumberType(
   const NumberType *values)> func;
```

- LLVM: Just in time (JIT) compilation to LLVM byte-code
  - Aggressive optimisation for vectorised expressions

## AD numbers as custom types

- Linearisation using AD instead of SD
- Wrappers for “lambda” optimiser, CSE
  - Taped AD types:
    Shared memory pool minimises active variable count

## LinearOperators as custom types

- Define solvers, preconditioners at run-time using a parameter file

```
// Create a symbolic expression to be evaluated
const SDNumberType symb_lo_A_inv("lo_A_inv");
const SDNumberType symb_la_u("la_u");
const SDNumberType symb_la_v("la_v");
const SDNumberType symb_la_y
  = symb_lo_A_inv * (symb_la_u + symb_la_v);

// Setup a linear system
SparseMatrix<number_t> la_A(...);
Vector<number_t>       la_u(...);
Vector<number_t>       la_v(...);

// Create linear operators
LinearOperator<Vector<number_t>> lo_A(la_A);
LinearOperator<Vector<number_t>> lo_A_inv
  = inverse_operator(lo_A, ...);

// Create a substitution map, linking the symbolic expression
// to the linear algebra
SDNumberType::substitution_map_t sub_vals;
sub_vals[symb_lo_A_inv] = SDNumberType(lo_A_inv, "A_inv");
sub_vals[symb_la_u]     = SDNumberType(la_u, "u");
sub_vals[symb_la_v]     = SDNumberType(la_v, "v");

// Substitution with evaluation
const Vector<number_t> la_y_result =
  static_cast<Vector<number_t>>(symb_la_y.substitution(sub_vals));
```

# Integration of SD into deal.II library

## Differentiation helpers

- Cell-level energy functional

$$\Pi(\boldsymbol{d}) \quad \rightarrow \quad \boldsymbol{R} = -\frac{d\,\Pi(\boldsymbol{d})}{d\boldsymbol{d}}, \boldsymbol{K} = \frac{d^2\Pi(\boldsymbol{d})}{d\boldsymbol{d} \otimes d\boldsymbol{d}}$$

- Cell-level residual linearisation

$$\boldsymbol{R}(\boldsymbol{d}) \quad \rightarrow \quad \boldsymbol{K} = -\frac{d\boldsymbol{R}(\boldsymbol{d})}{d\boldsymbol{d}}$$

```
template <int dim, typename NumberType>
class SDHelperVariationalFormulation : ... {
public:
  SDHelperVariationalFormulation(const unsigned int   &n_dofs_per_cell,
                                 const AdditionalData &additional_data);

  // Independent variables (symbolic)
  std::vector<sd_type> dof_values () const;
  sd_type JxW() const;

  template<typename FEValuesType, typename FEExtractorType>
  auto shape_function_gradients (const FEValuesType    &fe_values,
                                 const FEExtractorType &extractor,
                                 const std::string     &field_symbol);

  // Dependent variables (symbolic)
  void register_energy_functional (const sd_type &qp_energy, ...);

  // Optimiser (symbolic -> numeric)
  void optimize   (const typename sd_type::substitution_map_t &sub_vals);
  void substitute (const typename sd_type::substitution_map_t &sub_vals);

  // Local assembly (numeric)
  Vector<NumberType> compute_residual () const;
  FullMatrix<NumberType> compute_linearization () const;
}
```

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

# Symbolic differentiation example

## Variational formulation of quasi-static finite strain hyperelasticity

$$\Pi = \Pi^{int} + \Pi^{ext} \quad ; \quad \Pi^{int} = \int_{\mathcal{B}_0} \Psi_0 \left( \mathbf{C} \right) \, dV \quad ; \quad \min_{\mathbf{u}} \Pi \Rightarrow \delta \Pi = 0$$

# Integration of SD into deal.II library

## Creation of an SD helper class

```
void assemble_system_one_cell(...) const
{
typedef SD::SDHelperVariationalFormulation<dim, double> SDHelper;
typedef typename SDHelper::AdditionalData               SDHelper_AD;

// Initialise helper (one per combination of FE and material law)
static bool        initialised = false;
static SDHelper_AD sd_helper_ad(SD::OptimizerType::llvm);
static SDHelper    sd_helper(dofs_per_cell, sd_helper_ad);
```

# Integration of SD into deal.II library

## Defining the symbolic energy functional

```
// Configure helper
if (initialised == false)
{
  // Assume all QP's have same constitutive law
  const unsigned int q_point = 0;

  // Symbolic expression for kinematic quantities, in terms of DoF values
  const Tensor<2, dim, SDNumberType> fe_Grad_u =
    sd_helper.solution_gradient(scratch.fe_values, u_fe, "u");
const Tensor<2, dim, SDNumberType> fe_F =
  Physics::Elasticity::Kinematics::F(fe_Grad_u);
const SymmetricTensor<2, dim, SDNumberType> fe_C =
  Physics::Elasticity::Kinematics::C(fe_F);

// Symbolic expression for material free-energy
const SDNumberType psi = lqph[q_point]->get_symbolic_psi(fe_C);

// Symbolic expressions for finite element data
const SDNumberType JxW = sd_helper.JxW();

// Compute the cell energy contribution in terms of the field variables.
const SDNumberType energy_qp = psi * JxW;

// Register the dependent variables (i.e. the quadrature
// point contribution to the total cell energy)
sd_helper.register_energy_functional(energy_qp);
```

$$\nabla_0 \mathbf{u}$$

$$\mathbf{F} = \mathbf{I} + \nabla_0 \mathbf{u}$$

$$\mathbf{C} = \mathbf{F}^T \cdot \mathbf{F}$$

$$\Psi_0 \left( \mathbf{C} \right)$$

$$\int_{\mathcal{B}_0} \Psi_0 \left( \mathbf{C} \right) \, dV$$

# Integration of SD into deal.II library

## Defining the symbolic energy functional

```
// Configure helper
if (initialised == false)
{
  // Assume all QP's have same constitutive law
  const unsigned int q_point = 0;

  // Symbolic expression for kinematic quantities, in terms of DoF values
  const Tensor<2, dim, SDNumberType> fe_Grad_u =
    sd_helper.solution_gradient(scratch.fe_values, u_fe, "u");
```

$$\nabla_0 \mathbf{u}$$

```
const Tensor<2, dim, SDNumberType> fe_F =
  Physics::Elasticity::Kinematics::F(fe_Grad_u);
const SymmetricTensor<2, dim, SDNumberType> fe_C =
  Physics::Elasticity::Kinematics::C(fe_F);

// Symbolic expression for material free-energy
const SDNumberType psi = lqph[q_point]->get_symbolic_psi(fe_C);

// Symbolic expressions for finite element data
const SDNumberType JxW = sd_helper.JxW();

// Compute the cell energy contribution in terms of the field variables.
const SDNumberType energy_qp = psi * JxW;

// Register the dependent variables (i.e. the quadrature
// point contribution to the total cell energy)
sd_helper.register_energy_functional(energy_qp);
```

$$\mathbf{F} = \mathbf{I} + \nabla_0 \mathbf{u}$$

$$\mathbf{C} = \mathbf{F}^T \cdot \mathbf{F}$$

$$\Psi_0 \left( \mathbf{C} \right)$$

$$\int_{\mathcal{B}_0} \Psi_0 \left( \mathbf{C} \right) \, dV$$

# Integration of SD into deal.II library

## Invoking the optimiser

```
// Construct the symbolic optimisation map (inputs for optimiser)
typename SDNumberType::substitution_map_t sub_vals_optim;
const std::vector<Tensor<2, dim, SDNumberType>> fe_Grad_Nx_u =
  sd_helper.shape_function_gradients(scratch.fe_values, u_fe, "u");

SD::add_to_symbol_map(sub_vals_optim, sd_helper.JxW());
SD::add_to_symbol_map(sub_vals_optim, sd_helper.dof_values());
SD::add_to_symbol_map(sub_vals_optim, fe_Grad_Nx_u);

// Perform differentiation and optimisation of all of the resulting
// symbolic expressions
sd_helper.optimize(sub_vals_optim);

initialised = true;
}
```

# Integration of SD into deal.II library

## Invoking the optimiser

```
// Construct the symbolic optimisation map (inputs for optimiser)
typename SDNumberType::substitution_map_t sub_vals_optim;
const std::vector<Tensor<2, dim, SDNumberType>> fe_Grad_Nx_u =
  sd_helper.shape_function_gradients(scratch.fe_values, u_fe, "u");

SD::add_to_symbol_map(sub_vals_optim, sd_helper.JxW());
SD::add_to_symbol_map(sub_vals_optim, sd_helper.dof_values());
SD::add_to_symbol_map(sub_vals_optim, fe_Grad_Nx_u);

// Perform differentiation and optimisation of all of the resulting
// symbolic expressions
sd_helper.optimize(sub_vals_optim);

initialised = true;
}
```

# Integration of SD into deal.II library

## Assembly using precomputed derivatives and known data

```
// Extract the local degree-of-freedom values
// from the solution vector
std::vector<double> local_dof_values(dofs_per_cell);
for (unsigned int K = 0; K < dofs_per_cell; ++K)
  local_dof_values[K] = scratch.solution_total(data.local_dof_indices[K]);

// Update quadrature point solution
scratch.fe_values[u_fe].get_function_gradients(
  scratch.solution_total, scratch.solution_grads_u_total);

  // Use the optimiser to compute the residual and its
  // linearisation on this cell
  for (unsigned int q_point = 0; q_point < n_q_points; ++q_point)
  {
    // Perform symbolic substitution (symbols -> values)
    typename SDNumberType::substitution_map_t sub_vals;
    SD::add_to_symbol_value_map(sub_vals,
                                sd_helper.JxW(),
                                scratch.fe_values.JxW(q_point));
    // Same for DoF values, shape function gradients
    SD::add_to_symbol_value_map(...);
    sd_helper.substitute(sub_vals);

    // Compute the residual values and their Jacobian for the new
    // evaluation point
    data.cell_rhs -= sd_helper.compute_residual(); // RHS = - residual
    data.cell_matrix.add(1.0, sd_helper.compute_linearization());
  }
}
```

# Integration of SD into deal.II library

## Assembly using precomputed derivatives and known data

```
// Extract the local degree-of-freedom values
// from the solution vector
std::vector<double> local_dof_values(dofs_per_cell);
for (unsigned int K = 0; K < dofs_per_cell; ++K)
  local_dof_values[K] = scratch.solution_total(data.local_dof_indices[K]);

// Update quadrature point solution
scratch.fe_values[u_fe].get_function_gradients(
  scratch.solution_total, scratch.solution_grads_u_total);

  // Use the optimiser to compute the residual and its
  // linearisation on this cell
  for (unsigned int q_point = 0; q_point < n_q_points; ++q_point)
  {
    // Perform symbolic substitution (symbols -> values)
    typename SDNumberType::substitution_map_t sub_vals;
    SD::add_to_symbol_value_map(sub_vals,
                                sd_helper.JxW(),
                                scratch.fe_values.JxW(q_point));
    // Same for DoF values, shape function gradients
    SD::add_to_symbol_value_map(...);
    sd_helper.substitute(sub_vals);

    // Compute the residual values and their Jacobian for the new
    // evaluation point
    data.cell_rhs -= sd_helper.compute_residual(); // RHS = - residual
    data.cell_matrix.add(1.0, sd_helper.compute_linearization());
  }
}
```

# Integration of SD into deal.II library

## Assembly using precomputed derivatives and known data

```
// Extract the local degree-of-freedom values
// from the solution vector
std::vector<double> local_dof_values(dofs_per_cell);
for (unsigned int K = 0; K < dofs_per_cell; ++K)
  local_dof_values[K] = scratch.solution_total(data.local_dof_indices[K]);

// Update quadrature point solution
scratch.fe_values[u_fe].get_function_gradients(
  scratch.solution_total, scratch.solution_grads_u_total);

  // Use the optimiser to compute the residual and its
  // linearisation on this cell
  for (unsigned int q_point = 0; q_point < n_q_points; ++q_point)
  {
    // Perform symbolic substitution (symbols -> values)
    typename SDNumberType::substitution_map_t sub_vals;
    SD::add_to_symbol_value_map(sub_vals,
                                sd_helper.JxW(),
                                scratch.fe_values.JxW(q_point));
    // Same for DoF values, shape function gradients
    SD::add_to_symbol_value_map(...);
    sd_helper.substitute(sub_vals);

    // Compute the residual values and their Jacobian for the new
    // evaluation point
    data.cell_rhs -= sd_helper.compute_residual(); // RHS = - residual
    data.cell_matrix.add(1.0, sd_helper.compute_linearization());
  }
}
```

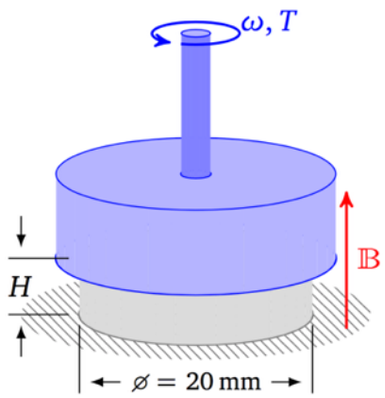
FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

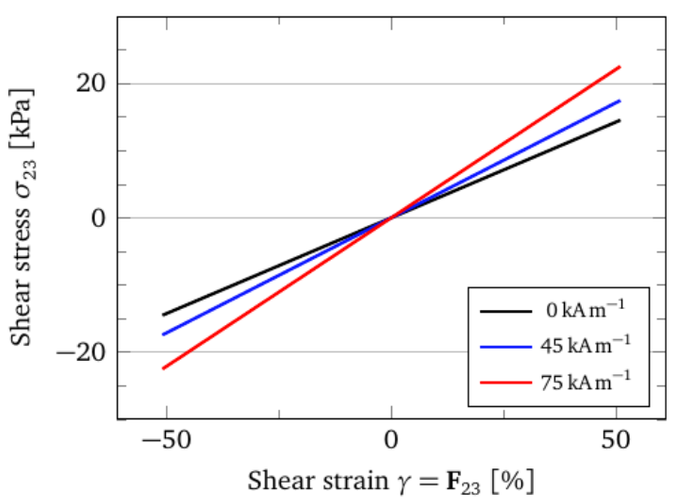
# Applications in magneto-mechanics

deal.II

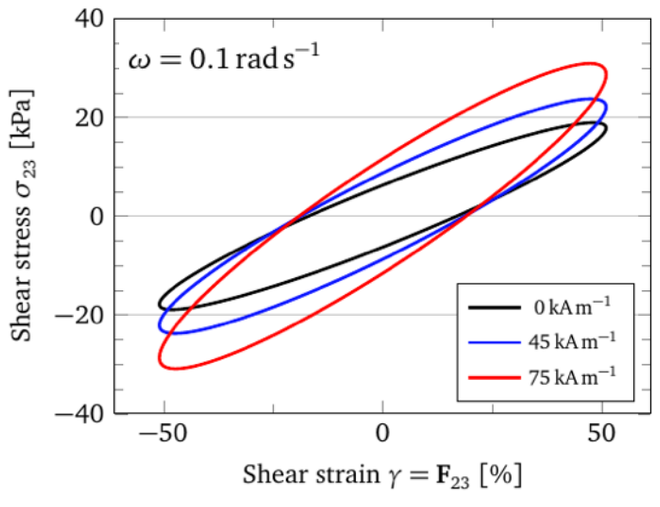
# Magneto-mechanics: Constitutive modelling

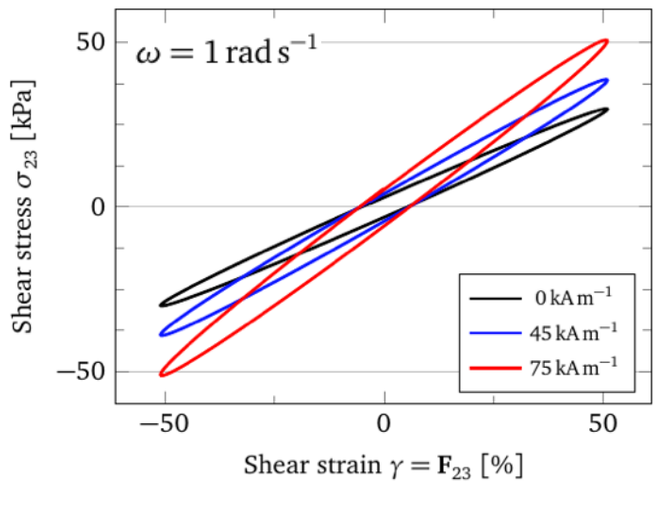



## Saturating magneto-elastic




## Saturating magneto-viscoelastic




## Thermodynamics

- Reduced dissipation inequality

$$\mathscr{D}_{\mathrm{int}} = \mathbf{S}^{\mathrm{tot}} : \frac{1}{2}\dot{\mathbf{C}} - \mathbb{B}\cdot\dot{\mathbb{H}} - \dot{\Psi}_0\left(\mathbf{C}, \mathbf{C}_{\mathrm{v}}^{i}, \mathbb{H}\right) \geq 0$$

- Total stress, magnetic induction

$$\mathbf{S}^{\mathrm{tot}} = 2\frac{\partial \Psi_0}{\partial \mathbf{C}} \quad , \quad \mathbb{B} = -\frac{\partial \Psi_0}{\partial \mathbb{H}}$$

- Decomposition of mixed free-energy/enthalpy

$$\Psi_0\left(\mathbf{C}, \mathbf{C}_{\mathrm{v}}^{i}, \mathbb{H}\right) = U_0\left(\mathbf{C}, \mathbf{C}_{\mathrm{v}}^{i}, \mathbb{H}\right) + \mu_{\mathrm{r}} M_0\left(\mathbf{C}, \mathbb{H}\right)$$

- Magnetic free energy

$$M_0\left(\mathbf{C}, \mathbb{H}\right) := -\frac{1}{2}\mu_0\left[\mathbb{H}\otimes\mathbb{H} : J\mathbf{C}^{-1}\right]$$

## Constitutive law

Pelteret (2018)

- Introduce split in magneto-mechanical energy

$$U_0\left(\mathbf{C}, \mathbf{C}_{\mathrm{v}}^{i}, \mathbb{H}\right) = \Psi_0^{\mathrm{vol}}(J) + f_{\mathrm{e}}^{\mathrm{sat}}(\mathbb{H})\,\overline{U}_0^{\mathrm{e}}\left(\overline{\mathbf{C}}\right) + f_{\mathrm{v}}^{\mathrm{sat}}(\mathbb{H})\,\overline{U}_0^{\mathrm{v}}\left(\overline{\mathbf{C}}, \mathbf{C}_{\mathrm{v}}^{i}\right)$$

- Volumetric energy

$$\Psi_0^{\mathrm{vol}}(J) = \frac{\kappa}{4}\left[J^2 - 1 - 2\log(J)\right]$$

- Saturation function

$$f_{\mathrm{e/v}}^{\mathrm{sat}}(\mathbb{H}) = 1 + \left[\frac{\mu_{\mathrm{e/v}}^{\infty}}{\mu_{\mathrm{e/v}}} - 1\right]\mathrm{erf}\left(\sqrt{\pi}\frac{\mathbb{H}\cdot\mathbb{H}}{|\mathbb{H}^{\mathrm{sat}}|^2}\right)$$

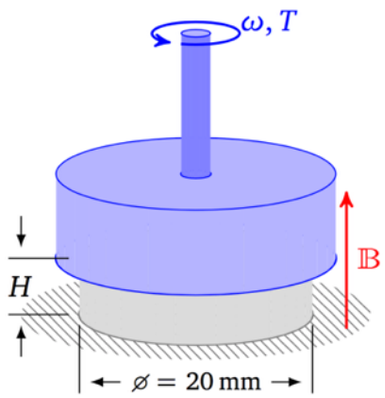
- Elastic energy

$$\overline{U}_0^{\mathrm{e}}\left(\overline{\mathbf{C}}, \mathbb{H}\right) = \frac{\mu_e}{2}\left[\overline{\mathbf{C}} : \mathbf{I} - \mathbf{I} : \mathbf{I}\right]$$

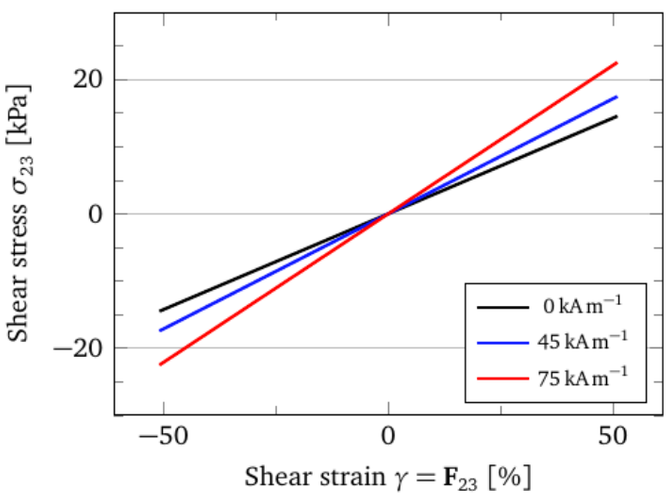
- Visco-elastic energy; evolution law

$$\overline{U}_0^{\mathrm{v}}\left(\overline{\mathbf{C}}, \mathbf{C}_{\mathrm{v}}^{i}, \mathbb{H}\right) = \frac{\mu_v}{2}\left[\mathbf{C}_{\mathrm{v}} : \overline{\mathbf{C}} - \mathbf{I} : \mathbf{I} - \log\left(\det\left(\mathbf{C}_{\mathrm{v}}\right)\right)\right] \qquad \dot{\mathbf{C}}_{\mathrm{v}} = \frac{1}{\tau_{\mathrm{v}}}\left[\overline{\mathbf{C}}^{-1} - \mathbf{C}_{\mathrm{v}}\right]$$

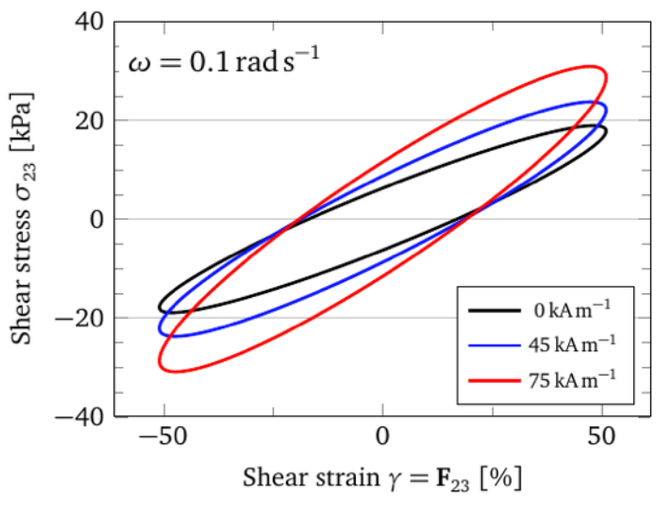
# Magneto-mechanics: Constitutive modelling

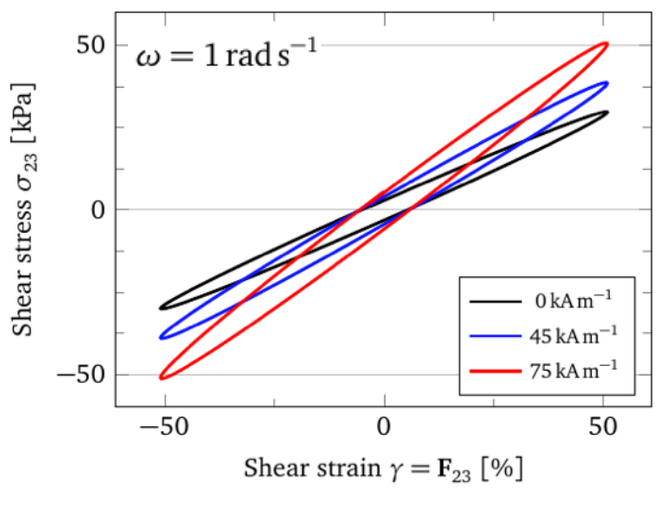



## Saturating magneto-elastic




## Saturating magneto-viscoelastic




## Thermodynamics

- Reduced dissipation inequality

$$\mathscr{D}_{\text{int}} = \mathbf{S}^{\text{tot}} : \frac{1}{2}\dot{\mathbf{C}} - \mathbb{B}\cdot\dot{\mathbb{H}} - \dot{\Psi}_0\left(\mathbf{C}, \mathbf{C}_{\text{v}}^{i}, \mathbb{H}\right) \geq 0$$

- Total stress, magnetic induction

$$\mathbf{S}^{\text{tot}} = 2\frac{\partial \Psi_0}{\partial \mathbf{C}} \quad , \quad \mathbb{B} = -\frac{\partial \Psi_0}{\partial \mathbb{H}}$$

- Decomposition of mixed free-energy/enthalpy

$$\Psi_0\left(\mathbf{C}, \mathbf{C}_{\text{v}}^{i}, \mathbb{H}\right) = U_0\left(\mathbf{C}, \mathbf{C}_{\text{v}}^{i}, \mathbb{H}\right) + \mu_{\text{r}} M_0\left(\mathbf{C}, \mathbb{H}\right)$$

- Magnetic free energy

$$M_0\left(\mathbf{C}, \mathbb{H}\right) := -\frac{1}{2}\mu_0\left[\mathbb{H}\otimes\mathbb{H} : J\mathbf{C}^{-1}\right]$$

## Constitutive law

Pelteret (2018)

- Introduce split in magneto-mechanical energy

$$U_0\left(\mathbf{C}, \mathbf{C}_{\text{v}}^{i}, \mathbb{H}\right) = \Psi_0^{\text{vol}}(J) + f_{\text{e}}^{\text{sat}}(\mathbb{H})\,\overline{U}_0^{\text{e}}\left(\overline{\mathbf{C}}\right) + f_{\text{v}}^{\text{sat}}(\mathbb{H})\,\overline{U}_0^{\text{v}}\left(\overline{\mathbf{C}}, \mathbf{C}_{\text{v}}^{i}\right)$$

- Volumetric energy

$$\Psi_0^{\text{vol}}(J) = \frac{\kappa}{4}\left[J^2 - 1 - 2\log(J)\right]$$

- Saturation function

$$f_{\text{e/v}}^{\text{sat}}(\mathbb{H}) = 1 + \left[\frac{\mu_{\text{e/v}}^{\infty}}{\mu_{\text{e/v}}} - 1\right]\text{erf}\left(\sqrt{\pi}\frac{\mathbb{H}\cdot\mathbb{H}}{|\mathbb{H}^{\text{sat}}|^2}\right)$$

- Elastic energy

$$\overline{U}_0^{\text{e}}\left(\overline{\mathbf{C}}, \mathbb{H}\right) = \frac{\mu_e}{2}\left[\overline{\mathbf{C}} : \mathbf{I} - \mathbf{I} : \mathbf{I}\right]$$

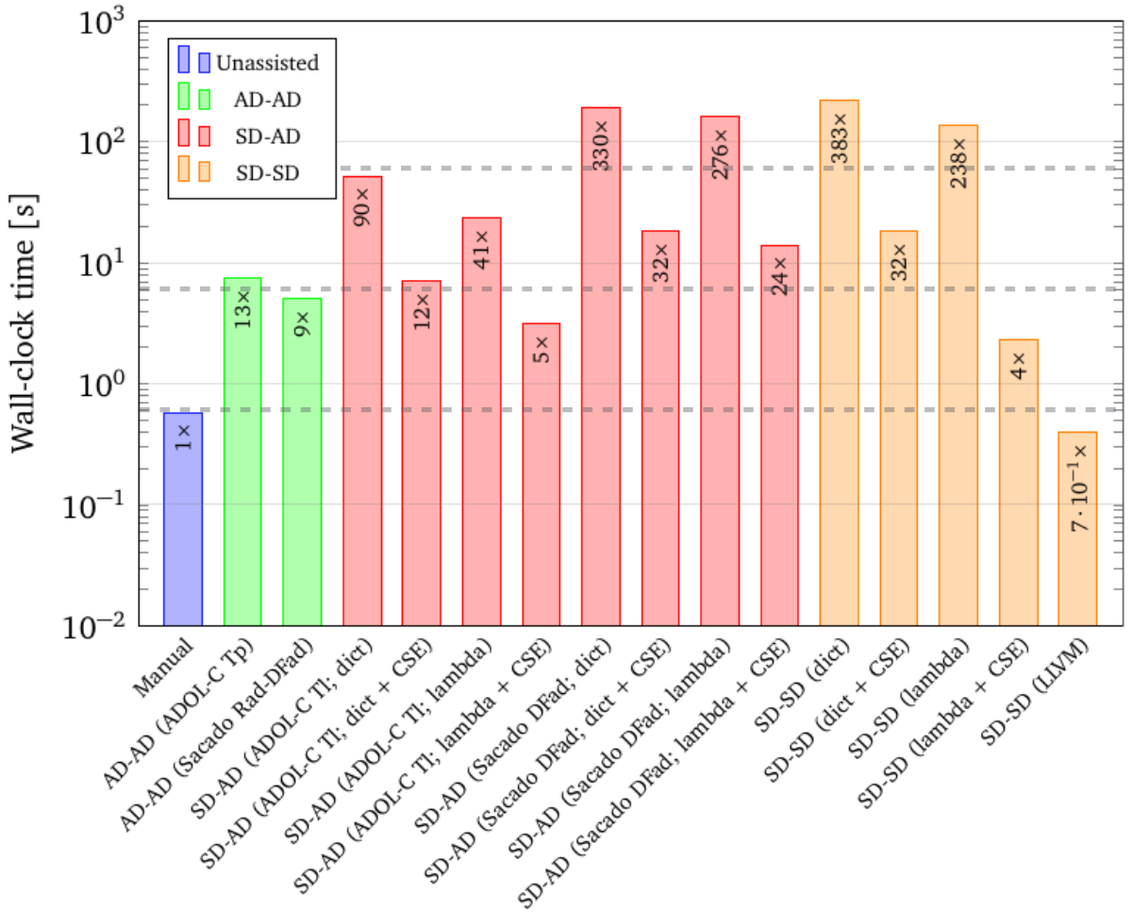
- Visco-elastic energy; evolution law

$$\overline{U}_0^{\text{v}}\left(\overline{\mathbf{C}}, \mathbf{C}_{\text{v}}^{i}, \mathbb{H}\right) = \frac{\mu_v}{2}\left[\mathbf{C}_{\text{v}} : \overline{\mathbf{C}} - \mathbf{I} : \mathbf{I} - \log\left(\det\left(\mathbf{C}_{\text{v}}\right)\right)\right] \qquad \dot{\overline{\mathbf{C}}}_{\text{v}} = \frac{1}{\tau_v}\left[\overline{\mathbf{C}}^{-1} - \mathbf{C}_{\text{v}}\right]$$

# Magneto-mechanics: Constitutive modelling

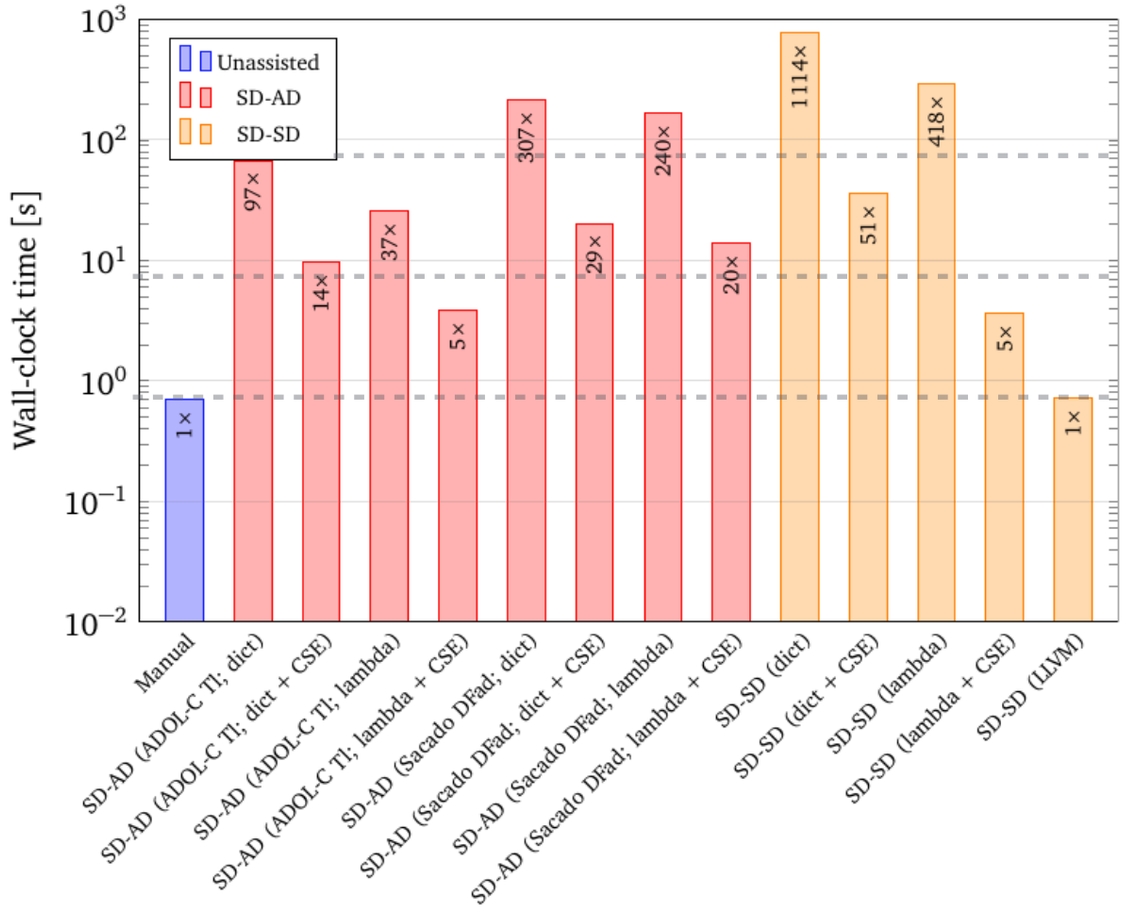
Timings for (repeated) simulated rheological experiment (computing all kinetic quantities + tangents)

## Saturating magneto-elastic




- **AD-AD** requires correction terms to work with dissipative materials
- **SD-AD**: Tapeless ADOL-C typically faster than Sacado DFad

## Saturating magneto-viscoelastic

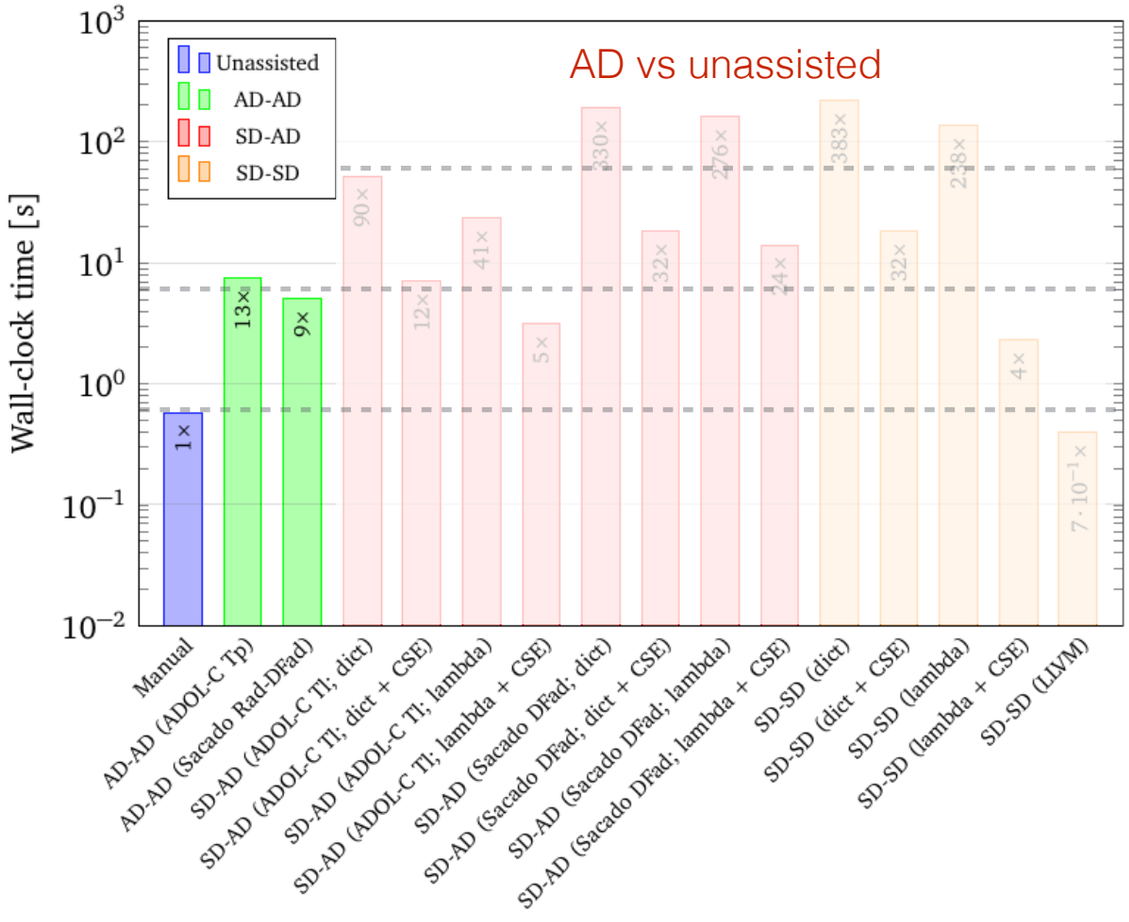



- **SD**: Lambda optimisation offers slight performance benefit in comparison to dictionary substitution
- **SD**: CSE provides large performance benefit
- **SD**: LLVM optimiser gives as good performance as hand implementation

# Magneto-mechanics: Constitutive modelling

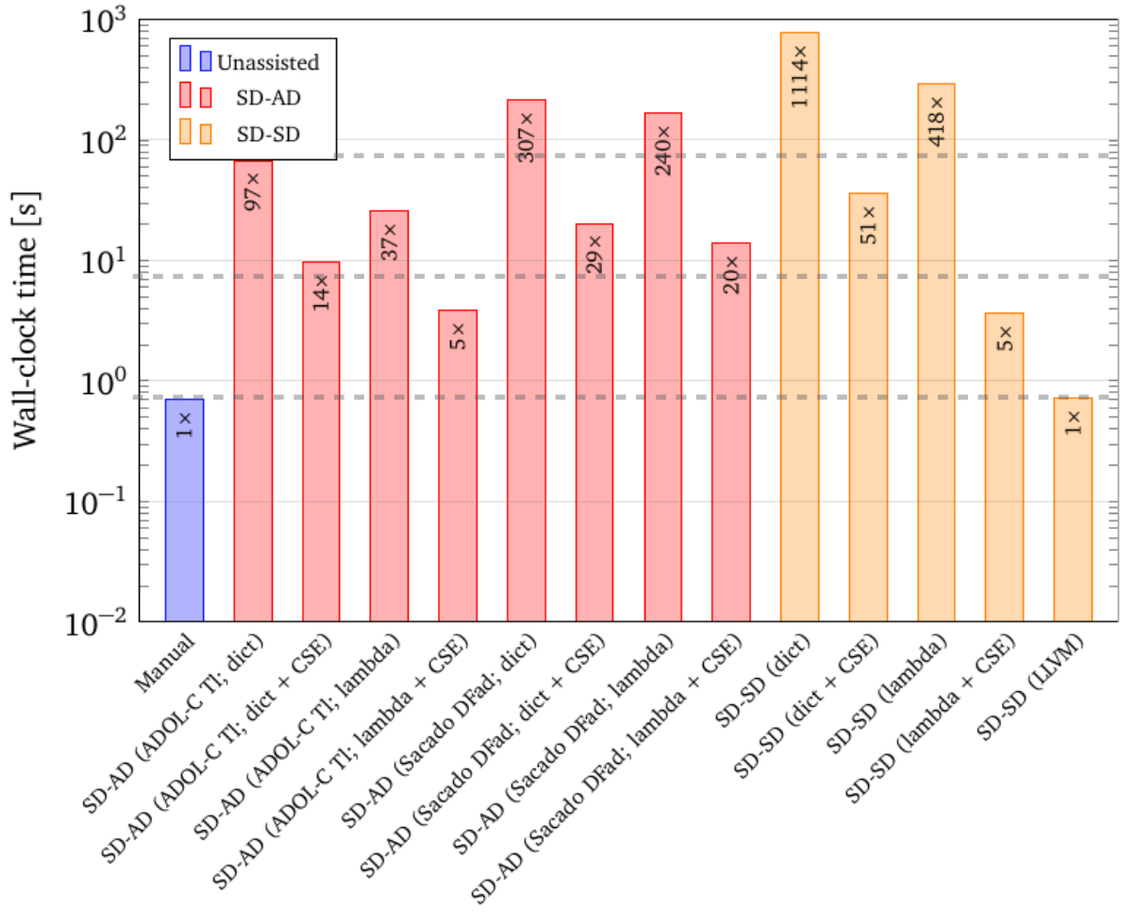
Timings for (repeated) simulated rheological experiment (computing all kinetic quantities + tangents)

## Saturating magneto-elastic




- **AD-AD** requires correction terms to work with dissipative materials
- **SD-AD**: Tapeless ADOL-C typically faster than Sacado DFad

## Saturating magneto-viscoelastic

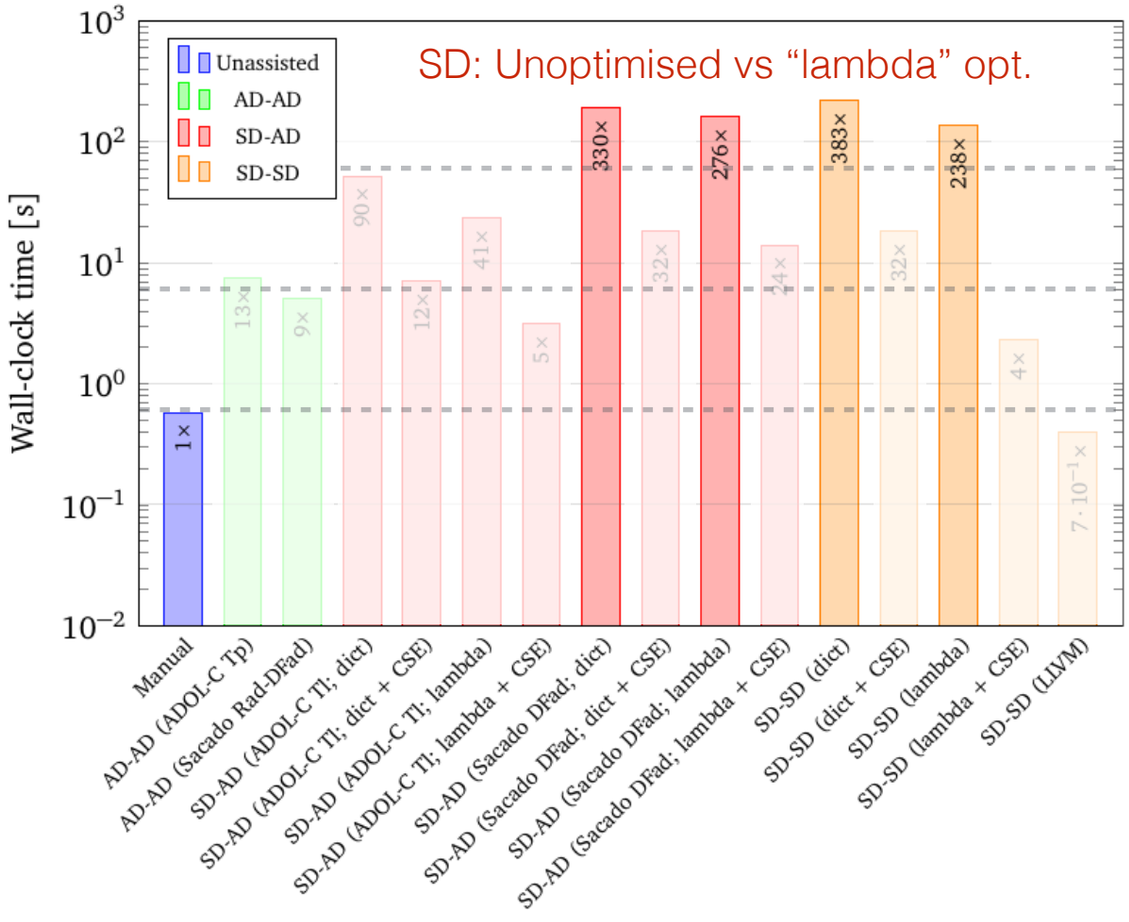



- **SD**: Lambda optimisation offers slight performance benefit in comparison to dictionary substitution
- **SD**: CSE provides large performance benefit
- **SD**: LLVM optimiser gives as good performance as hand implementation

# Magneto-mechanics: Constitutive modelling

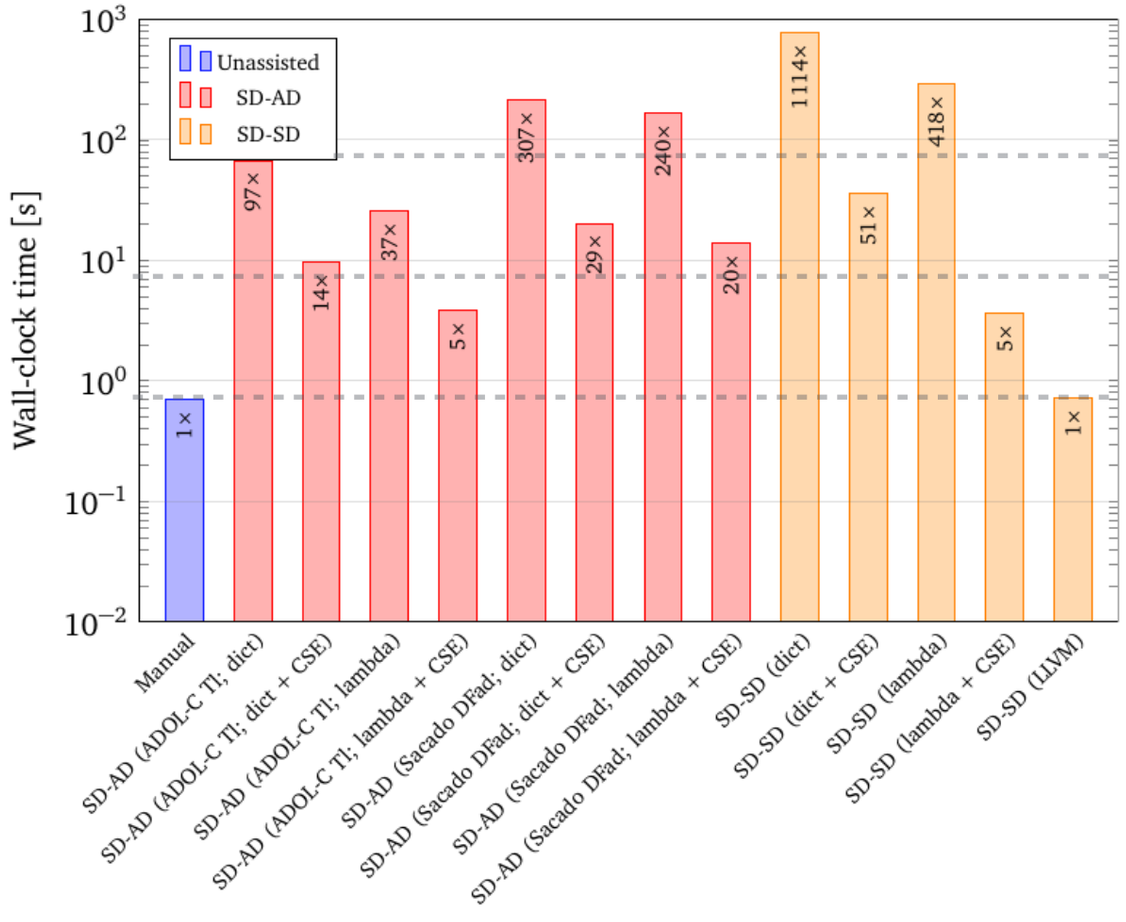
Timings for (repeated) simulated rheological experiment (computing all kinetic quantities + tangents)

## Saturating magneto-elastic




- **AD-AD** requires correction terms to work with dissipative materials
- **SD-AD**: Tapeless ADOL-C typically faster than Sacado DFad
- **SD**: Compiler influences unoptimised results

## Saturating magneto-viscoelastic

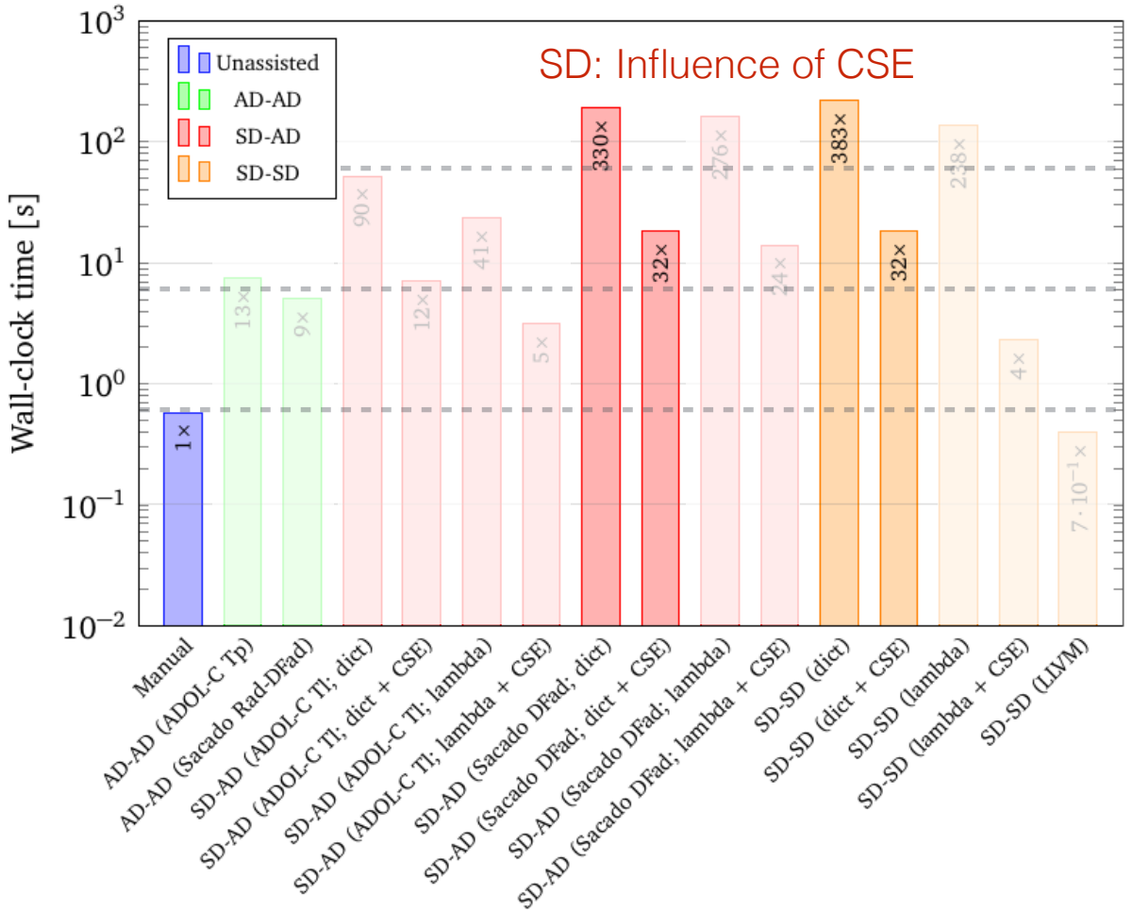



- **SD**: Lambda optimisation offers slight performance benefit in comparison to dictionary substitution
- **SD**: CSE provides large performance benefit
- **SD**: LLVM optimiser gives as good performance as hand implementation

# Magneto-mechanics: Constitutive modelling

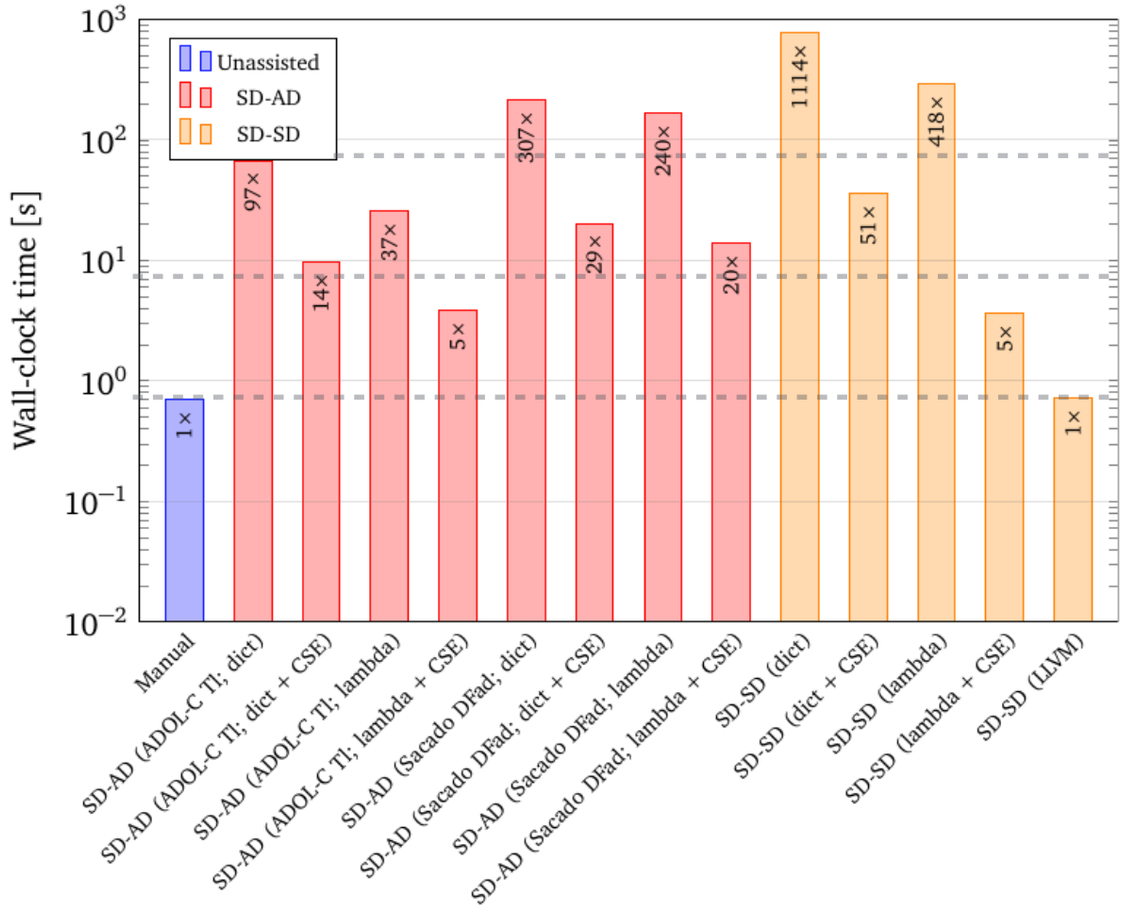
Timings for (repeated) simulated rheological experiment (computing all kinetic quantities + tangents)

## Saturating magneto-elastic




- **AD-AD** requires correction terms to work with dissipative materials
- **SD-AD**: Tapeless ADOL-C typically faster than Sacado DFad

## Saturating magneto-viscoelastic

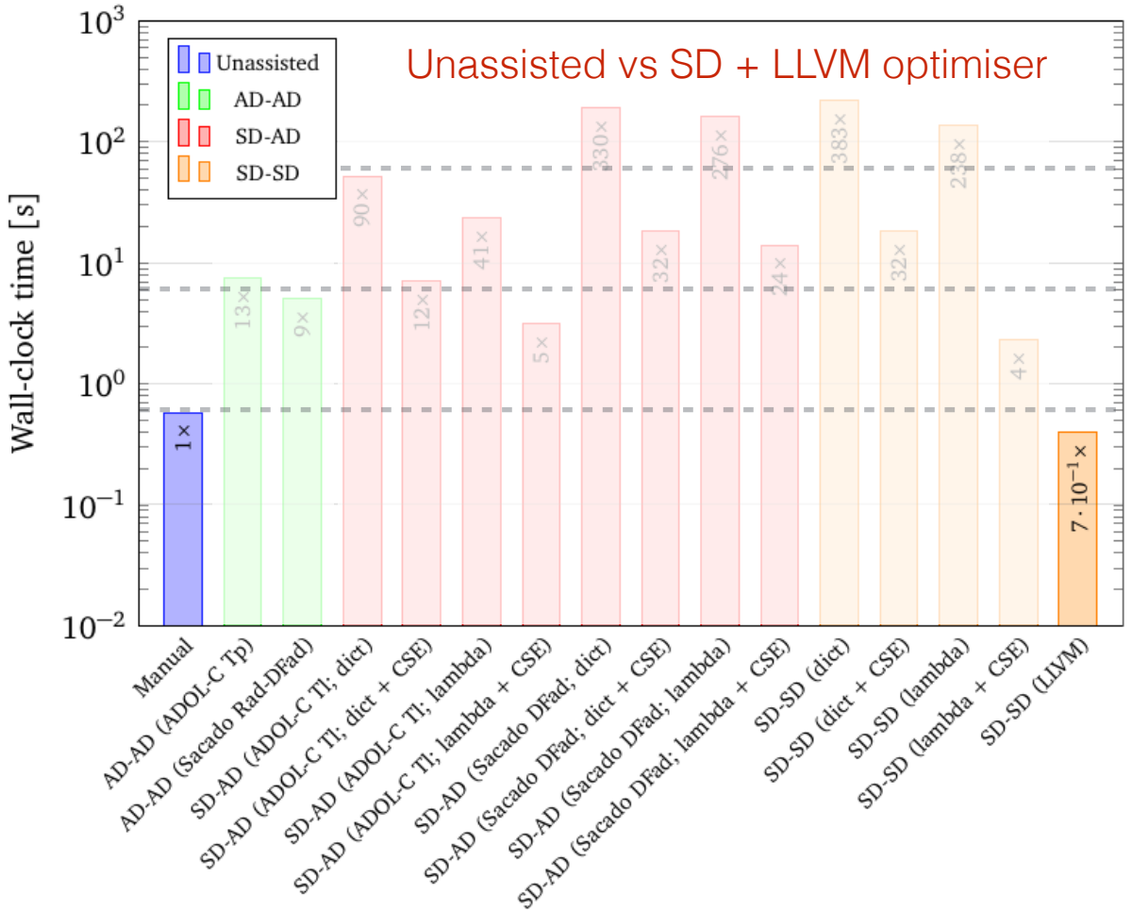



- **SD**: Lambda optimisation offers slight performance benefit in comparison to dictionary substitution
- **SD**: CSE provides large performance benefit
- **SD**: LLVM optimiser gives as good performance as hand implementation

# Magneto-mechanics: Constitutive modelling

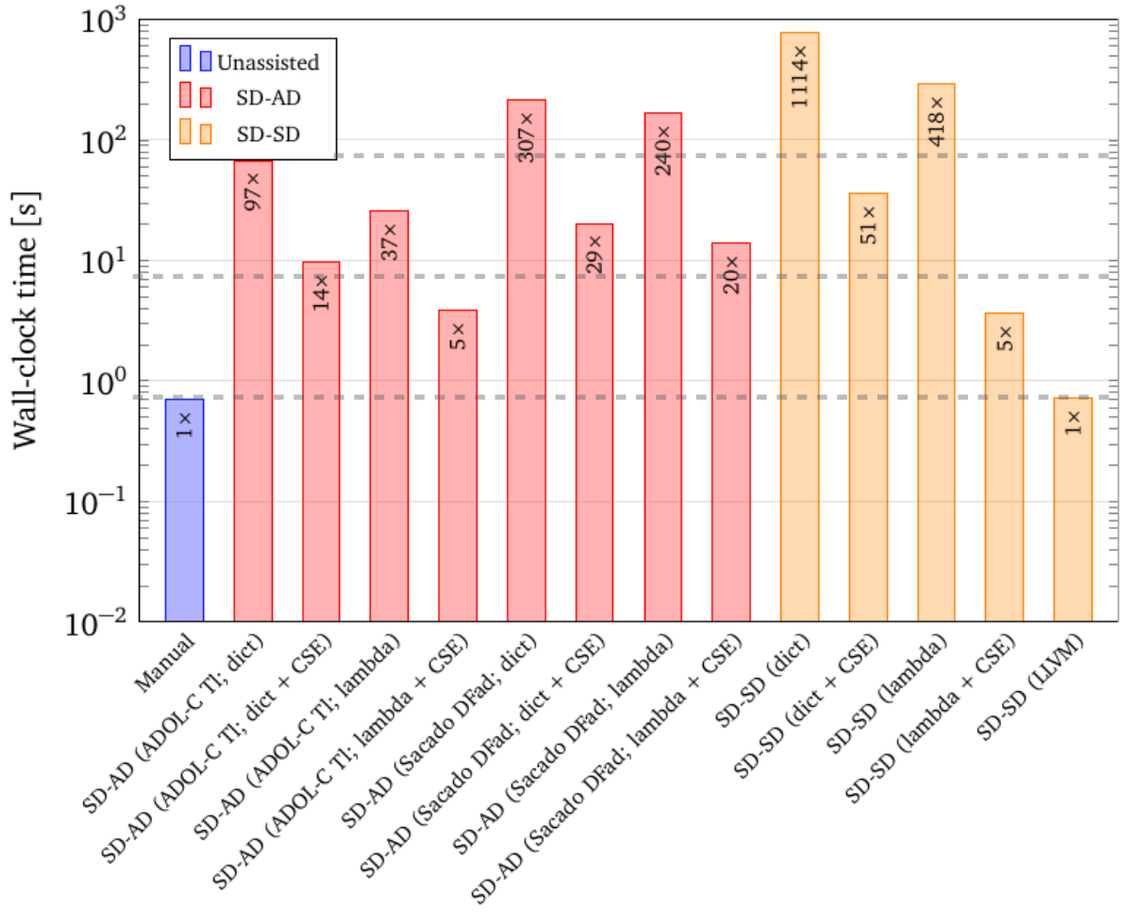
Timings for (repeated) simulated rheological experiment (computing all kinetic quantities + tangents)

## Saturating magneto-elastic




- **AD-AD** requires correction terms to work with dissipative materials
- **SD-AD**: Tapeless ADOL-C typically faster than Sacado DFad

## Saturating magneto-viscoelastic




- **SD**: Lambda optimisation offers slight performance benefit in comparison to dictionary substitution
- **SD**: CSE provides large performance benefit
- **SD**: LLVM optimiser gives as good performance as hand implementation

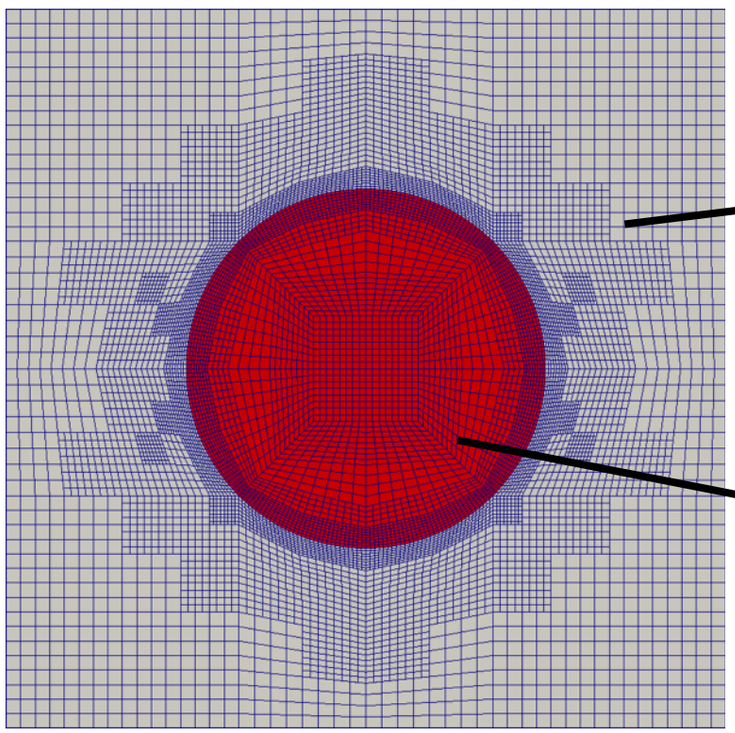
# Magneto-mechanics: FEM model of an RVE

## Problem configuration

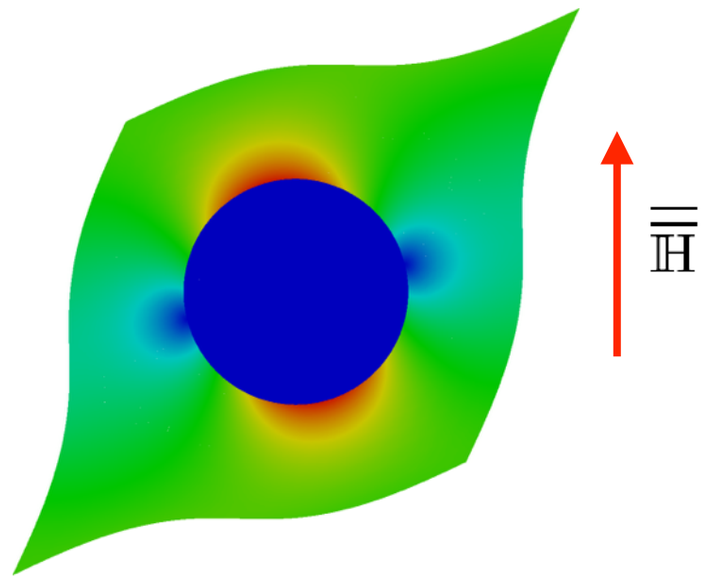
- Variational formulation: $\Pi^{int} = \int_{\mathcal{B}_0} \Psi_0\left(\mathbf{C}, \mathbb{H}\right)\, dV \quad ; \quad \min_{\boldsymbol{\varphi}} \max_{\phi} \Pi \Rightarrow \delta\Pi = 0$
- Weak formulation: $\int_{\mathscr{B}_0} \delta\mathbf{E} : \mathbf{S}^{\mathrm{tot}}\, \mathrm{dV} = \int_{\partial\mathscr{B}_0^{\mathbf{t}}} \delta\boldsymbol{\varphi} \cdot \mathbf{t}_0^{\mathrm{mech}}\, \mathrm{dA} + \int_{\mathscr{B}_0} \delta\boldsymbol{\varphi} \cdot \mathbf{b}_0^{\mathrm{mech}}\, \mathrm{dV} \qquad \int_{\mathscr{B}_0} \boldsymbol{\nabla}_0 \delta\Phi \cdot \mathbb{B}\, \mathrm{dV} = \int_{\partial\mathscr{B}_0^{\mathbb{B}}} \delta\Phi\, \left[\mathbf{N} \cdot \mathbb{B}\right]\, \mathrm{dA}$
- Material model: NeoHookean + spatially linear magneto-elastic contribution
- Number of DoFs: 155 000

- Nested iterative solver (CG + AMG preconditioner)
- 24 MPI processes
- DoF-based load balancing

## Discretised geometry




## Magnetic field in deformed configuration



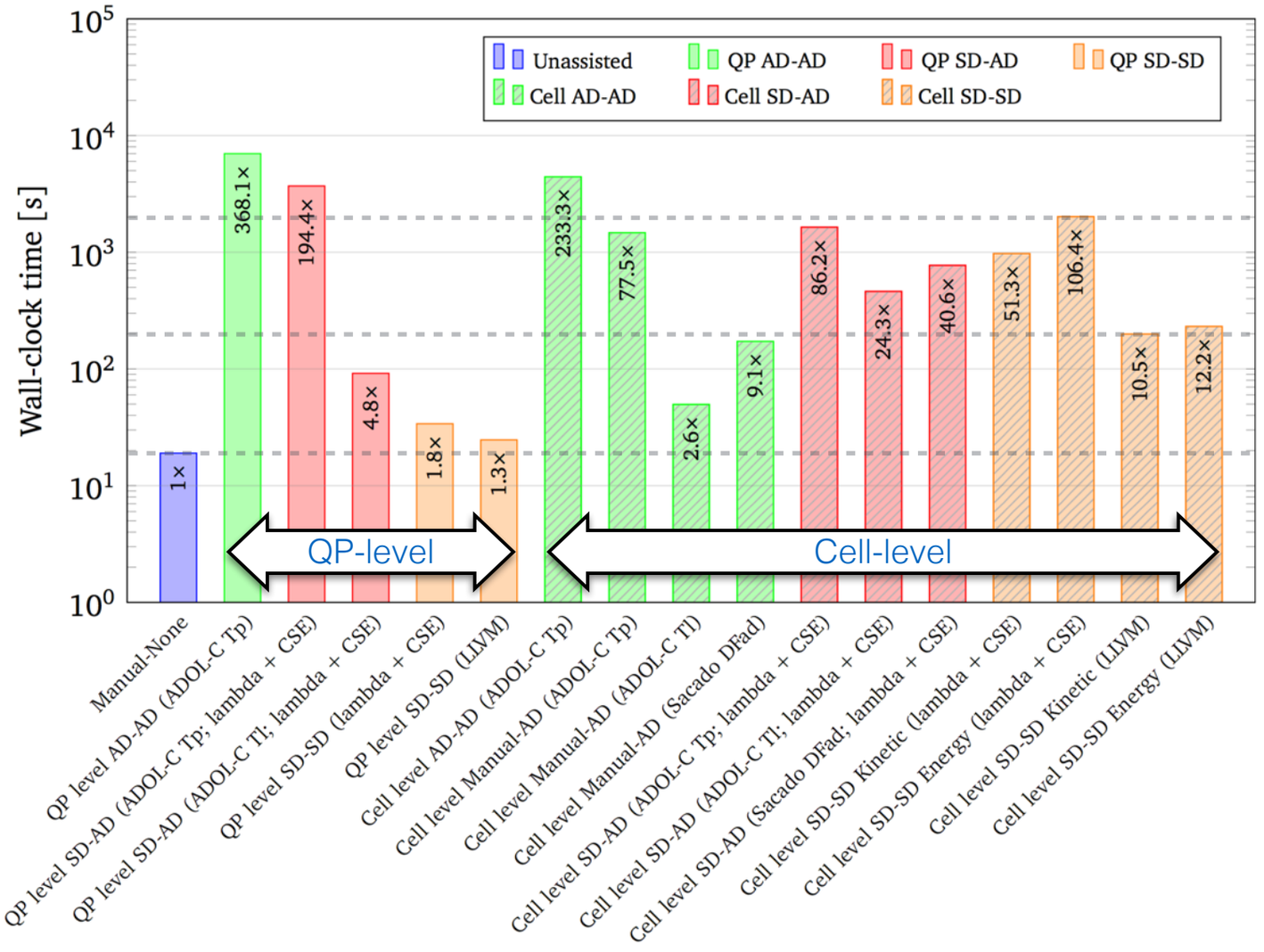

$$\overline{\overline{\mathbf{F}}} = \begin{pmatrix} 1 & 0.25 \\ 0.25 & 1 \end{pmatrix} \quad , \quad \overline{\overline{\mathbb{H}}} = \begin{bmatrix} 0 \\ 225 \end{bmatrix} \mathrm{kA\,m^{-1}}$$

# Magneto-mechanics: FEM model of an RVE

## Efficiency during assembly

- Taped **AD** very slow
  - Limit re-taping
- Residual linearisation using tapeless **AD** quite efficient
- Applying **SD** at QP-level is very efficient
- Cell-level **SD** + LLVM optimiser ~10x slower than hard-coding
- Tapeless **SD-AD** + optimisations results in intermediate performance

Assembly time

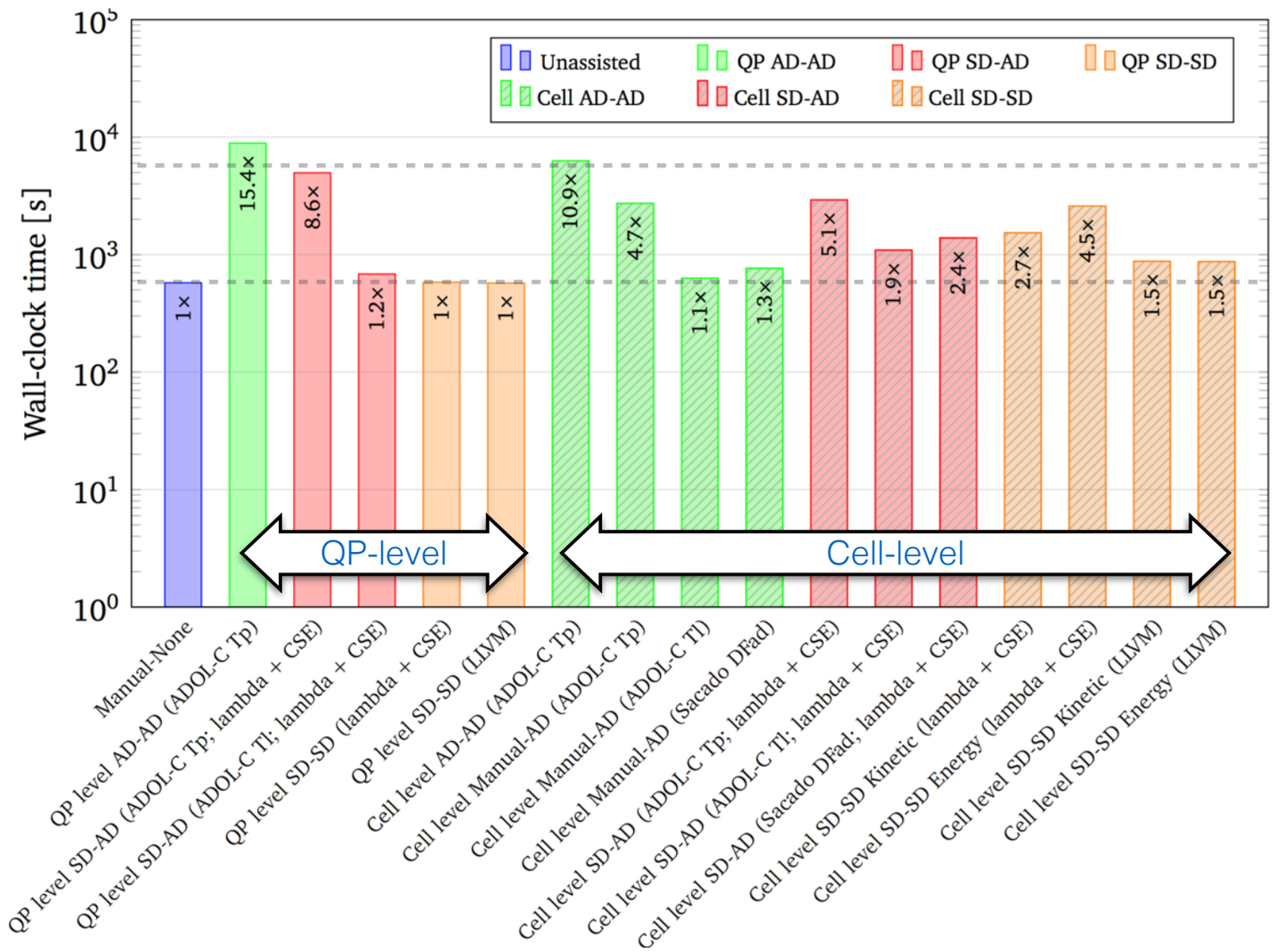


# Magneto-mechanics: FEM model of an RVE

## Overall efficiency

- Most efficient methods have ~0-50% runtime penalty
- Less performance penalty as number of DoFs per core is reduced
- Overhead even less visible if using a direct solver
- Caution: Results influenced by:
  - Stiffness of linear system
  - Constitutive model
  - Parallelisation
  - …

Total simulation time



# Summary and outlook

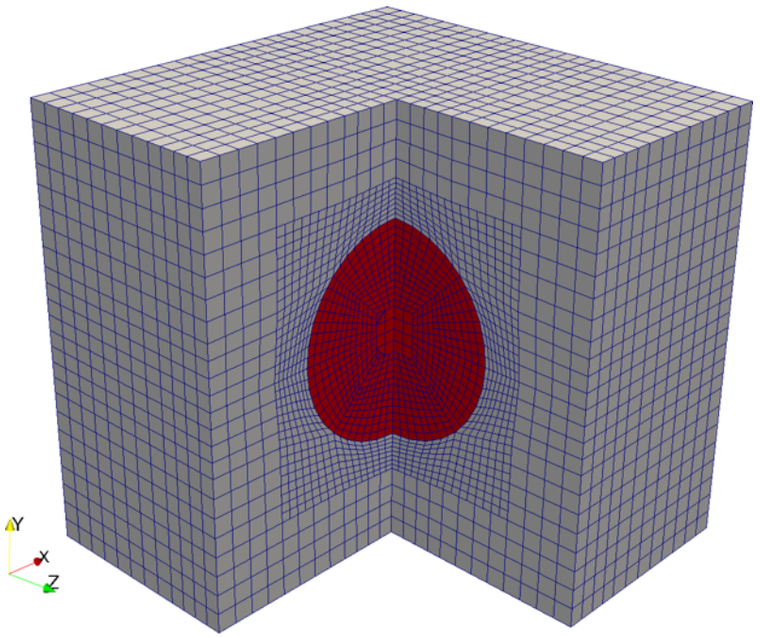
## Observations and conclusions

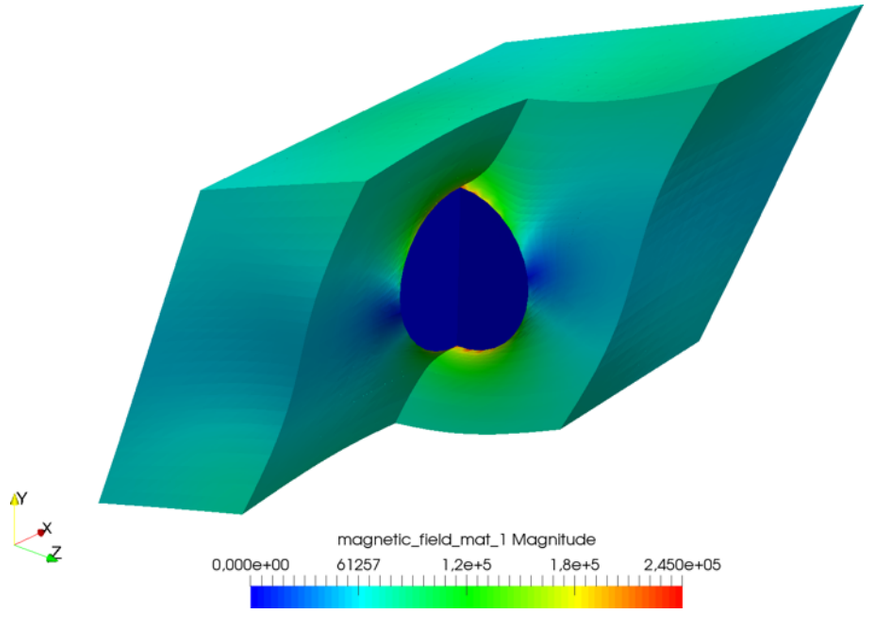
- Leveraging **AD**/**SD** reduces the number of equations to be implemented
  - Finite element assembly & constitutive modelling
- Choosing a suitable **AD** library / implementation is not necessarily trivial
- Enhances ability to rapid prototype
  - May require a “paradigm shift” (**SD**)
- Non-negligible performance loss when using **AD** or **SD**
  - FEM: Most efficient **AD**: Cell-level residual linearisation (ADOL-C tapeless)
  - FEM: Most efficient **SD**: Quadrature-point description of constitutive law (with LLVM optimisation)
- Easiest framework to use: **AD** (almost drop-in replacement; with some caveats)
- Most robust framework: Pure **SD**

# Summary and outlook

## Future work

- (Re-)Perform parametric FEM study in 3d
- Investigate other AD libraries
- Operator-overloaded C++ wrapper for SymEngine
  - Extend as new (FEM-related) functionality is added
  - Upstream to the SymEngine
- Investigate exotic uses of SD library



# Thank you for your attention


ERC Advanced Investigator Grant – **MOCOPOLY**
Multi-Scale, Multi-Physics, **MO**delling and
**CO**mputation of magneto-sensitive **POLY**meric materials

## Libraries and frameworks

- deal.II
- ADOL-C
- Trilinos
- SymEngine
- METIS

# References


Pelteret, J.-P. V.; Fernando, I.; McBride, A. & Steinmann, P.
*Implementation of a symbolic and automatic differentiation framework in deal.II*
In preparation, 2018

Alzetta G.; Arndt, D.; Bangerth, W.; Boddu, V. ; Brands, B.; et al.
*The `deal.II` Library, Version 9.0*
Journal of Numerical Mathematics, 2018, Accepted

Pelteret, J.-P. V.; Walter, B. & Steinmann, P.
*Application of metaheuristic algorithms to the identification of nonlinear magneto-viscoelastic constitutive parameters*
Journal of Magnetism and Magnetic Materials, 2018, In press

Pelteret, J.-P. V.; Davydov, D.; McBride, A.; Vu, D. K. & Steinmann, P.
*Computational electro-elasticity and magneto-elasticity for quasi-incompressible media immersed in free space*
International Journal for Numerical Methods in Engineering, Wiley-Blackwell, 2016, 108, 1307-1342

Pelteret, J-P. V.
*A computational neuromuscular model of the human upper airway with application to the study of obstructive sleep apnoea* PhD Thesis, University of Cape Town, 2013

de Borst, R. & van der Giess, E.
*Material Instabilities in Solids*
John Wiley & Sons, 1998. ISBN 978-0-471-97460-4

Fike, J. and Alonso, J.
*The Development of Hyper-Dual Numbers for Exact Second-Derivative Calculations*
49th Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition, 2011, 886, 124–141

Gay, D.
*Mixed Integer Nonlinear Programming: Using expression graphs in optimization algorithms*
Springer, 2012, 247-262

Griewank, A; Juedes, D. & Utke, J.
*Algorithm 755: ADOL-C: a package for the automatic differentiation of algorithms written in C/C++*
ACM Transactions on Mathematical Software, 1996, 22, 131–167

Hagopian, M.
*Contraction bands at short sarcomere length in chick muscle*
Journal of Cell Biology, 1970 ,47, 790–796

Meurer, A.; Smith, C. P. ; Paprocki, M.; Čertík, O.; Kirpichev, S. B.; et al.
*SymPy: symbolic computing in Python*
PeerJ Computer Science, 2017, 3, pp. e103,

Miehe, C.; Vallicotti, D. & Zäh, D.
*Computational structural and material stability analysis in finite electro-elasto-statics of electro-active materials*
International Journal for Numerical Methods in Engineering, 2015, 102, 1605–1637

Phipps, E. & Pawlowski, R.
*Recent Advances in Algorithmic Differentiation: Efficient Expression Templates for Operator Overloading-based Automatic Differentiation*
Springer, 2012, 73, 309-319

Rudykh, S. & Bertoldi, K.
*Stability of anisotropic magnetorheological elastomers in finite deformations: A micromechanical approach*
Journal of the Mechanics and Physics of Solids, 2013, 61, 949–967

Rudykh, S., Bhattacharya, K. & deBotton, G.
*Multiscale instabilities in soft heterogeneous dielectric elastomers*
Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences, 2013, 470, 20130618--20130618